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

水口大知　　渡邊 宏

産業技術総合研究所　システム検証研究センター

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*

水口大知[†]　渡邊 宏[‡]
産業技術総合研究所 システム検証研究センター[§]

概 要

システム開発における効率的な不具合発見の方法として，モデル検査 (model checking) が注目を集めている．本稿では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を紹介し，そこで得られた知見を報告する．そして設計段階での仕様書に対するモデル検査の適用の有効性を示すとともに，問題点や課題を考察する．

1　はじめに

システム開発における効率的な不具合発見の方法として，数理的技法を用いたシステム検証技術が注目を集めている [6]．中でも 80 年代初めに提案され今日までに理論的な成熟を遂げたモデル検査 (model checking)[3] は，それを実装した様々なツールが公開されており [1]，ハードウェアやプロトコルの設計では既に実用化段階にある．しかしながら，組み込みソフトウェアのモデル検査事例は少なく，その実用性は明らかではない．そこで本稿は，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を報告し，その有効性を示す．また，現状での問題点や課題を指摘する．

2　モデル検査

モデル検査とは有限な状態遷移系が，時相論理式で表された性質を満足するかどうかを判定する数理的手法のことである．モデル検査ツールはこれを計算機上で自動的に実行し，真偽値を返す．特に結果が偽の場合には，性質を満足しない状態遷移列も返す．これを反例と呼ぶ．反例は状態遷移系としてモデル化されたシステムが望ましくない動作をすることの証拠であり，不具合発見の手がかりとなる．

モデル検査の大きな特徴は，与えられた性質が成り立つかどうかの判定をする際に，システムのとりうる全状態を網羅的に探索することである．このことは実行列を予め考えてから，それを 1 つ 1 つ検査しなければならないテストと大きく異なる．またこれにより従来の不具合検出手法よりも精密な検査が可能であるとされる．

しかしながらこの網羅的な探索のために，計算には多くのメモリを消費する．実際にツールを使用すると，計算がメモリの限度を超える，あるいは終了しないことがよく起こる．これを避けるにはモデルの状態数を減らしてサイズを小さくしなければならないが，あまりに小さなモデルでは十分な検査が出来ない．検証の精度を保ちつつサイズの小さなモデルを作成することが，モデル検査の実適用における成功の鍵であり最大の困難である．

3　検査対象

仕様書に潜む不備を発見することは，製品の障害を未然に防ぐ上で重要である．そこで仕様書に対してモデル検査を適用し，不備を発見することを目指す試行実験を行った．

本事例では，企業で実際に開発途中にある製品のソフトウェア仕様書を検査対象とした．この製品はメイン処理および割り込み処理によって機能を実現する組み込み機器で，比較的小規模なもの

*A Case Study of Applying Model Checking to Embedded Software Development

[†]Daichi Mizuguchi. daichi.mizuguchi@aist.go.jp
[‡]Hiroshi Watanabe. hirowata@ni.aist.go.jp
[§]Research Center for Verification and Semantics (CVS), National Institute of Advanced Industrial Science and Technology (AIST). http://unit.aist.go.jp/cvs

である．そのうち主な 4 つの機能についてモデル検査を実施した．

ソフトウェア仕様書は，複数のモジュール仕様書から構成されている．モジュール仕様書は C 言語によるコーディング作業に直接用いられるもので，モジュール毎に詳細なフロー図が記載されたものである．

このソフトウェア仕様書に対してモデル検査を適用したところ，いくつかの不備を指摘することができた．

以下，作業手順の概略を第 4 節に述べ，詳細を第 5 節から第 10 節において順次述べる．第 11 節においてまとめを述べる．なお作業体制については文献 [8] を参照されたい．

4 作業手順

今回実施した仕様書に対するモデル検査の作業のあらすじを以下に示す．

4.1 モデル検査用仕様書の作成

作業はモデル検査用の仕様書を作成することから始まった．既に述べたが，今回は実際にプログラミングに用いられるソフトウェア仕様書を検査対象とした．しかしながら，これを直接モデル検査に用いることは様々な理由から困難であったため，専用の仕様書を作成することにした．詳細は第 5 節に述べる．

4.2 モデル化および抽象化

次にモデル検査用仕様書から，この組み込みシステムの状態を規定する変数およびその値の変化の条件を抽出し，これに基づいて各変数の変化の様子をモデル検査ツール NuSMV[2] の言語にコード化した．

このとき，モデル検査用仕様書に書かれた通りの素朴なモデル化では，状態数が大きすぎてツールの実行が終了せず，検査結果を得ることができなかった．そこで抽象化作業を行いモデルのサイズを小さくする必要があった．

第 6 節においてモデル化の詳細を述べ，抽象化作業の内容を第 7 節に述べる．

4.3 検査項目の設定と検査式の作成

モデル化の作業と並行して，モデル検査用仕様書もしくは検査対象に関する事前の知識に基づき，この組み込みシステムに期待される性質を列挙して，検査項目とした．それらを時相論理 CTL (Computation Tree Logic)[3, 5] の論理式に翻訳して検査式を作成した．

本事例で使った検査項目は，設定の目的により大きく 2 種類に分けられる．すなわち，モデルの妥当性検査を目的とするものと，ソフトウェアの動作検査を目的とするものの 2 種類である．システムのモデル化が適切であることは本来目的とするソフトウェア部分の検査の前提であるから，まず始めにこれを検査する必要があった．これをモデルの妥当性検査と呼ぶことにする．

モデルの妥当性検査について第 8 節に述べ，ソフトウェアの動作検査について第 9 節に述べる．

4.4 ツールの実行と反例解析

モデルと検査式が出来たのち，双方をモデル検査ツールに入力して検査を実行した．モデル検査ツールは，入力されたモデルが検査式を満足するかどうかを自動的に計算して結果を回答する．

モデルの妥当性検査に対する結果は，すべてが真となることを要請した．妥当性検査の結果が偽である場合には，モデル化が不適切であると判断し，結果が真となるようにモデルを修正した．

妥当性検査をすべて真とするモデルが得られた後，仕様書に対するソフトウェアの動作検査を行った．この段階で，いくつかの検査式に対して検査結果が偽になったため，得られた反例を解析して原因究明に当たった．そのうちのいくつかは仕様書の不備が原因であることが判明した．これを開発現場にフィードバックすることで，設計の改善に貢献することができた．

一方，妥当性検査では検出できなかったモデル化の誤りが原因と判断される場合や，検査項目や

検査式の間違いが原因と判断される場合もあった．これらの場合は，誤り修正して再度ツールを実行した．

反例の解析は当初の予想よりも時間と労力のかかる作業であった．第 10 節に詳細を述べる．

5　モデル検査用仕様書の作成

5.1　モデル検査用仕様書の必要性

今回モデル検査の適用対象としたソフトウェア仕様書は，さらに複数のモジュール仕様書から構成されている．モジュール仕様書にはモジュール毎に詳細なフロー図が記載されており，それを基に C 言語によるコーディング作業が行われる．1 つのモジュールは，およそ 1 つの関数に対応する．

モデル検査適用の目的は，機能仕様の内容がソフトウェア仕様書に記載されたアルゴリズムによって実現されるかどうかを確かめることである．

しかしながら，実際のソフトウェア仕様書に直接モデル検査を適用することは困難であったため，専用の仕様書を作成することにした．その理由を以下に列挙する．

- ソフトウェア仕様書が大きすぎたこと．
- 機能とモジュールの対応を明確にする必要があったこと．
- 各変数が取り得る値を明確にする必要があったこと．
- ハードウェアや外部要因もモデル化する必要があったこと．

これらについて，仕様書の概要とともに以下に詳しく述べる．

5.2　ソフトウェアモジュールの仕様書

モデル検査を適用するとき，システムの仕様全体を 1 つのモデルにより記述し検査できるならば，それが最も理想的であろう．しかしながら，今回のソフトウェア仕様書の大きさと現状のツールの能力を考慮すれば，それが不可能であることは明らかであった．そこで対象とする製品の様々な機能のうち，主要な 4 つに注目して検査を実施することとした．そのため，機能別に仕様書を用意することにした．

ところが，設計の便宜上，ソフトウェア仕様書は必ずしも上で述べた 4 つの機能別に分割されているわけではなかった．例えば，割り込み処理用のモジュールにおいては，異なる箇所で異なる機能のための処理が記載されることがよくあった．そのため，機能毎にモデルを作成して検査項目を設定するためには，機能毎に関連するモジュールを集約して再構成した仕様書を用意する必要があった．

また，実際のソフトウェア仕様書には変数毎にそのサイズのみが記載されていたが，新たに作成した仕様書には変数が取り得る値の範囲も加えて明記した．次のモデル化の作業において必要となるためである．

ソフトウェアの各モジュールはメイン処理を構成するものと，割り込み処理として呼び出されるものの 2 種類に分けられる．前者は，メインの無限ループを実現するメインループモジュールと，それから巡回的に呼び出されて処理を行ういくつかのメイン処理モジュールに分かれる．後者はハードウェアからの割り込み要求により随時起動される処理（割り込み処理モジュール）である．

5.3　ハードウェアモジュールの仕様書

通常の組み込み機器がそうであるように，今回対象としたシステムのソフトウェアもハードウェアの状態や動作と密接に連携して一連の処理を行う．よってソフトウェアの動作を検査するには，ハードウェアという動作環境が必要となる．このようなとき，モデル検査ではソフトウェアのモデルに加えて，ハードウェアのモデルも作成して検査すればよい．

そのため今回は，検査のために必要なハードウェアの動作を抜粋して記載した仕様書も新たに作成した．一般に組み込み開発の現場ではこのような仕様書は存在せず，技術者は必要に応じて CPU のマニュアルなどを参照するのが普通であろう．本事例ではこの新たな仕様書を参照することにより，ハードウェアに精通していない者でもモデル化の

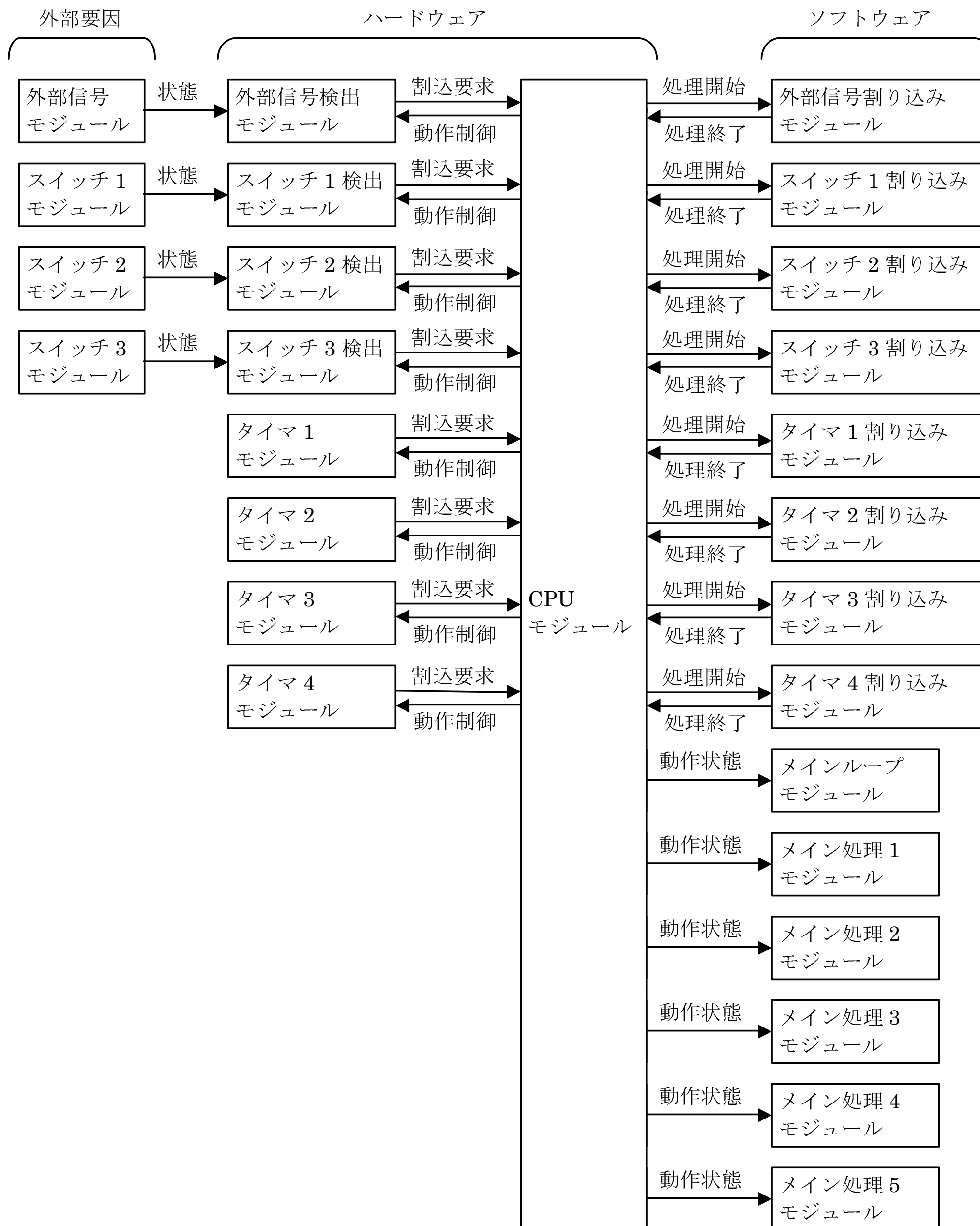

図 1. モジュールとその依存関係

表 1. 機能とモジュール

機能	ソフトウェアモジュール	ハードウェアモジュール	外部要因モジュール
A	外部信号割り込み タイマ 1,2 割り込み	CPU，外部信号検出 タイマ 1,2	外部信号
B	外部信号割り込み タイマ 1,2 割り込み メインループ メイン処理 1	CPU，外部信号検出 タイマ 1,2	外部信号
C	外部信号割り込み スイッチ 1,2 割り込み タイマ 1,2,3,4 割り込み メインループ メイン処理 1,2,3,4,5	CPU，外部信号検出 スイッチ 1,2 検出 タイマ 1,2,3,4	外部信号 スイッチ 1,2
D	外部信号割り込み スイッチ 1,2,3 割り込み タイマ 1,2,3 割り込み メインループ	CPU，外部信号検出 スイッチ 1,2,3 検出 タイマ 1,2,3	外部信号 スイッチ 1,2,3

作業に従事できた．

今回モデル化を必要としたハードウェアの機能は，以下の 3 つであった．

- 外部要因の変化を検出し割り込み要求を出力する機能．
- 所定の周期毎に割り込み要求を出力するタイマ機能．
- 割り込み要求を検出し，優先順位などを踏まえて所定のソフトウェアモジュールを起動させるとともに，割り込み要求元の動作を開始させたり停止させたりする機能．

これらのハードウェアの各機能のこともモジュールと呼ぶことにし，それぞれ (外部要因) 検出モジュール，タイマモジュール，および，CPU モジュールと名付ける．

5.4　外部要因モジュールの仕様書

外部要因はハードウェアの動作に影響し，ハードウェアの動作はソフトウェアの動作に影響を与える．モデル検査では，外部要因の振る舞いもモデル化することで，外乱などを含めた様々な環境下でソフトウェアの動作を検査できる．

しかしながら，仕様書の中に外部要因についての詳細を記載することはまれである．そこで今回は，検査に必要となるいくつかの外部要因の特性を新たに仕様書に明記し，これに基づいてモデル化を行った．

今回モデル化を必要とした外部要因は，次の 2 つであった．

- ある連続値をもつ外部信号の変化．
- いくつかのスイッチの状態変化．

それぞれの外部要因を，外部信号モジュール，および，スイッチモジュールと呼ぶことにする．

5.5　機能とモジュール

モデル検査用仕様書に記載された全てのモジュールとその間の依存関係を図 1 に示す．

今回検査対象とした 4 つの機能を A，B，C，D と呼ぶことにしよう．それぞれの機能がどのモジュールから構成されるかを表 1 に示す．

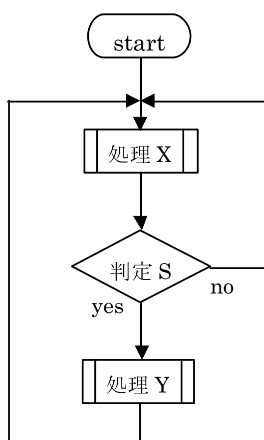

図 2. フローチャート例

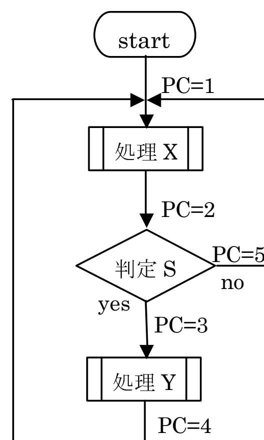

図 3. プログラムカウンタの設定例

機能 A は，外部信号および 2 つのタイマによる割り込み処理によって実現される．機能 B は，機能 A と 1 つのメイン処理モジュール（メイン処理 1 モジュール）を組み合わせることによって実現される．機能 C および D は，機能 A にいくつかのメイン処理モジュールと割り込み処理モジュールを加えることによって実現される．

ちなみに，機能 A は外部信号の状態を監視するための機能で，外部信号が正常から異常へ変化したことの検出，およびその逆の変化の検出を行う．そのために 2 つのタイマを組み合わせて用いている．これはこの組み込み機器の最も基本となる機能の 1 つである．また機能 B は，機能 A で捉えた外部信号の変化に従って，CPU の動作状態を適切に変化させるための機能である．機能 C は機器の内部状態をモニタを通して表示するための機能であり，機能 D は機器に取り付けられたスイッチを通じて，機器への操作を行うための機能である．

6　モデル化

6.1　ソフトウェアモジュールのモデル化

ソフトウェア部分は仕様書に記載されたフロー図を基にモデル化を行った．フロー図から状態遷移系を作るには，システムの状態を規定する変数に加えて，フローにおける位置を表す変数（プログラムカウンタ）を導入すればよい．

プログラムカウンタを用いたフロー図のモデル化の手順は以下のとおりである．まず，フローの各処理および条件分岐の出入り口において，プログラムカウンタの値を設定する．次にプログラムカウンタの変化の条件を記述する．最後に，各処理の開始条件をプログラムカウンタに連動して記述する．

簡単な例を以下に示す．図 2 に示すフローチャートのモデル化を考えよう．このフローのプログラムカウンタを PC として，モデル化を行う．まず，PC を図 3 に示すように設定する．次に，PC の変化を表 2 のように定める．ただし，初期値は 1 とする．さらに，PC = 1 のときに処理 X を，PC = 3 のときに処理 Y をそれぞれ開始すると定める．このようにすれば，図 2 の処理をモデル化することができる．

6.2　ハードウェアモジュールのモデル化

ハードウェアモジュールのモデル化においては，特に以下の 3 点に留意した．

1 つ目は，割り込み要求の優先順位である．例えば外部信号検出による割り込み要求の優先順位は，タイマ 2 のそれよりも高く設定されていた．つまり，それらの割り込み要求が同時に発生した場合，外部信号割り込みモジュールを先に起動し，その終了を待ってタイマ 2 割り込みモジュールを起動することがハードウェアの役割となっていた．

表 2. プログラムカウンタの変化条件

今の PC	変化の条件	次の PC
1	処理 X が終了した	2
2	判定 S が真	3
2	判定 S が偽	5
3	処理 Y が終了した	4
4 または 5	なし	1

2 つ目は，多重割り込みの禁止である．例えば，タイマ 1 割り込みモジュールの実行中にタイマ 2 の割り込み要求が発生した場合，その要求はタイマ 1 割り込みモジュールの処理が終了するまでハードウェアによって保留される．

3 つ目は，メイン処理と割り込み処理の排他性である．対象のシステムにおいては，通常はメイン処理モジュールが逐次起動されるが，割り込み要求発生時には随時割り込み処理によって中断される (図 4 参照)．従って例えばメイン処理 1 モジュールの実行中にタイマ 1 による割り込みが発生した場合は，直ちにタイマ 1 割り込みモジュールが起動されるとともにメイン処理 1 モジュールの処理は一旦停止する．タイマ 1 割り込みモジュールの処理が終了すると，メイン処理 1 モジュールの処理が再開される．ただしメイン処理においては，割り込み禁止区間を設定することができる．割り込み禁止区間の実行中に割り込み要求が発生した場合，その要求は割り込み禁止区間内の処理が終了するまで保留される．

以上の動作がすべて実現されるようにハードウェアをモデル化した．

6.3 外部要因モジュールのモデル化

外部信号は，異常時には不規則に発生する上に，正常時でも一定のゆらぎがある．それを模擬するために，非決定的に動作するモデルを作成した．

またスイッチについても任意のタイミングで状態が変化する可能性があるので，基本的には非決定的にスイッチが入ったり切れたりするモデルを作成した．

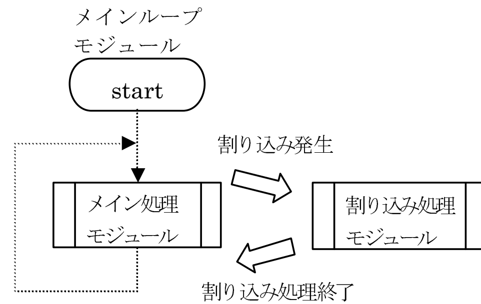

図 4. 割り込み処理の流れ

6.4 単位時間の設定

このシステムがタイマを複数含むことなどから分かるように，モデル化においては時間の概念を取り入れなければならなかった．ところが NuSMV では時間を連続値のまま直接取り扱うことができない．

そこで今回は，1 回の状態遷移にある一定の時間が経過するという方針で時間を離散化することにより，モデルに時間の概念を取り入れた．この方針に基づき，それぞれのタイマに対しては，起動から経過した時間を記録する変数を 1 つずつ割り当て，タイマの動作を模擬した．また，割り込み処理やメイン処理に要する時間についても，モデルにおいて各処理の開始から終了までに要する状態遷移数（ステップ数）を調整することで考慮した．

時間概念の取り入れ方については，他の手法も考えられるだろう．例えば今回は，システムの動作開始からの時間をカウントするための時計変数を特別に用意し，時間の経過とプログラムの処理を完全に分離して取り扱う方法も試してみた．しかしながらこの手法では，各処理ごとに時間変数をカウントアップする必要があるためモデル化の作業が煩雑であった．また，時計変数の変域を大きくとる必要があるため，状態数の増大を避け難かった．そこで先に示した，モデル化が比較的容易であり時計変数が必要ない方針を採用した．

7　抽象化

7.1　2 段階の抽象化

本事例において，当初は仕様書にできるだけ忠実なモデル化を試みた．だがそれではモデルの状態数があまりに大きくなりすぎて手元のツールおよび計算機環境では計算が完了せず，検査結果を得ることが出来なかった．そこでいくつかの抽象化を行いモデルの状態数の削減を試みた．

今回行った抽象化は大きく 2 種類に分けられる．すなわち仕様書レベルでの抽象化，および，NuSMV のモデルレベルでの抽象化である．以下それぞれについて順に説明する．

7.2　仕様書レベルでの抽象化

仕様書に対しては，以下の抽象化を施して簡略化を行った．

1. 変数の取捨選択
   仕様書の中で検査に関係する変数を選び出し，関係しない変数と，それに対する処理を仕様書から除いた．検査式には直接現れていない変数であっても，検査結果に影響を与える場合には残した．
   この手法は，プログラムスライシングや cone of influence reduction 手法 [3, 4] に分類される．ただし今回は理論的に厳密な適用はしておらず，システムを熟知している設計者が，知識と経験に基づく判断により行った．
2. 変数の定義域の有限化
   有限なモデルを生成するために，各変数の定義域を有限状態とした．例えば，時刻の変数は仕様書の中で実数型が割り当ててあったが，モデルの上では 0 からある定数 n までの自然数を周期的に動くものとした．また，本来は連続的な物理量である外部信号も，有限な離散値により表現した．
3. 変数の縮約
   検査項目に出てくるシステムの性質が，仕様書の中で複数の変数の値の組み合わせで表現される場合，それらの変数を 1 つの変数で表現をした．
   例えば，いくつかの検査項目の中でタイマ 1 の動作状態を参照したが，この状態はシステム内部では 2 つのブール変数 (x_1 と x_2 としよう) によって表されていた．そこでタイマ 1 の動作状態を表す変数を 1 つ用意して (y としよう)，x_1 と x_2 に関する処理を y に関する処理で置き換えた．このとき，タイマ 1 の動作状態は停止か作動の 2 つのみであったため，y もブール変数とすることができた．これによりモデルのサイズを削減できた．
4. モジュールの取捨選択
   検査の精度を大きく損なうことなく動作の簡略化が可能であると判断した場合，不要となるモジュールを削除した．

7.3　モデルレベルでの抽象化

上記の抽象化を施した仕様書に基づきモデルを作成したが，さらにモデルの状態数を削減する必要があった．そこで以下に示す作業を行った．

1. プログラムカウンタの削減
   フロー図においては数回のステップにより記述されている複数の処理に対して，可能と判断される場合には，これらをまとめて 1 つのステップとしてモデル化した．これにより，必要なプログラムカウンタの変域を削減することができ，その結果モデルの状態数を削減できた．
2. モデルの単位時間の調整
   第 6 節で設定した単位時間の調整を行った．最初は 1 回の遷移あたり 0.1 ミリ秒程度の微視的なモデルを作成した．しかしこのモデルでは検査が終了しなかったため，単位時間を 100 ミリ秒程度まで徐々に粗くして行きながら，検査が可能な単位時間を探した．
3. 非決定的動作の削減
   非決定的な動作は，モデルの状態数を増大させる大きな要因である．そこでモデルの非決定的動作を減らす抽象化を行った．これを特に外部要因に対して行った．例えばスイッチモジュールは，当初は完全に非決定的な動作によりモデル化していた．しかし，これでは

表 3. モデルの大きさ

機能	変数の個数 (2 値変数のみで表した場合)	NuSMV コードの行数	モデルの状態数
C	73 個 (93 個)	約 2800 行	約 3000 万状態
D	70 個 (87 個)	約 1500 行	約 2000 万状態

検査に時間がかかりすぎたため，いくらか変化に制限を設けたモデルも作成して併用した．これについては 9.1 項も参照されたい．

4. 割り込み可能箇所の削減
   メイン処理においては，割り込み禁止区間を除き，随時割り込み処理による中断の可能性がある．当初はこれに忠実なモデルを作成したが，可能と判断される場合には，割り込み可能な箇所を削減し，これによりモデルの状態数を削減した．

以上の抽象化作業により得られた機能 C および D のモデルについて，大きさに関するデータを表 3 に示す．

7.4 不備検出のための抽象化

一般に，モデル検査のための抽象化では，抽象化前のモデルと抽象化後のモデルの間に遷移系の模倣もしくは双模倣といった数学的関係があることが望ましい．模倣や双模倣が ACTL や CTL*などの時相論理式の充足性を保存するという命題 [3] を利用して，抽象化後のモデルで成り立つ性質は抽象化前のモデルでも成り立つという論法が使えるからである．

今回の抽象化は上に述べた手法を，検査可能なサイズのモデルが得られるまで適宜適用したものであった．そのため，抽象化前と後のモデルの間に模倣や双模倣などの関係があることは期待できず，抽象化によって保存される性質についても数学的な議論ができない．よってそもそも今回の作業を「抽象化」と呼ぶことについては，意見が分かれるところかもしれない．

だが今回はあくまで「不備検出のためのモデル検査」を目的としたので，抽象化の数学的厳密性についてはあまりこだわらないこととした．この目的のためには早い段階で検査可能なサイズのモデルを得て，できるだけ早くモデル検査作業に入ることが重要だからである．検査式を入力しても結果が得られないモデルでは，不備の検出はおろか，モデルが妥当であるかどうかの判断すらできない．今回，抽象化されたモデルに対する検査の結果，反例として出力された不審な挙動は，仕様書の不備を発見する大きな手掛りになった．従って,「不備検出のためのモデル検査」という観点からすれば，今回の抽象化は適切なものであったと考える．

ところで今回，抽象化の厳密性について全く考慮しなかったわけではない．既に 4.3 項で触れたが，抽象化後のモデルには妥当性検査を実施した．つまりソフトウェアの動作検査に先だって，モデルが最低限持つべき性質を検査項目としていくつか列挙し，抽象化後のモデルがそれらを満足することは確認した．これにより，抽象化したモデルの妥当性の確保に努めた．次節にて詳細を述べる．

8 モデルの妥当性検査

今回実施したモデルの妥当性検査の例を以下に列挙する．

1. 各割り込み要求が発生することがあること．
2. 複数の割り込み要求が同時に発生することがあること．
3. 複数の割り込み要求が同時に発生した場合，優先順位に従って受け付けられること．
4. 割り込み処理モジュールが動作しているときには，メイン処理モジュールは進行しないこと．
5. ソフトウェアモジュールの各処理が実行されることがあること．
6. 割り込み要求が上書きされることがないこと．

表 4. 割り込み要求や外部要因変化の発生タイミングの実現方法

No.	方法
1	任意の箇所で非決定的に与える
2	発生が可能な箇所を制限した上で，非決定的に与える
3	周期を定める変数を用い，周期的に与える
4	周期を定める変数を用い周期的に与えるが，周期に微少な変化を与える
5	No. 1 で，1 回のメインループ処理内での発生回数に制限を設ける

1 から 4 はハードウェアモジュールのモデル化が妥当であることの検査である．

5 はモデルの中にいわゆるデッドコードが無いことの検査で，ソフトウェア部分のモデル化が適切かどうかの検査である．もちろん仕様書の不備が原因でこの検査に通らないこともある．しかしながら，ある程度の査読を経た仕様書の場合はむしろモデル化の誤りが原因であることが多く，その修正のためにこの検査が役立つ．

6 は，モデルの抽象化により検査の必要が生じた項目である．このシステムにおいて，ある割り込み要求が発生してから対応する割り込み処理が起動されるまでには，時間差が生じることがある．割込要求発生時に，他の割り込み処理が実行されている場合やメイン処理が割り込み禁止区間内にある場合である．だが割り込み要求が長時間待たされたのでは，期待される動作を損なう可能性がある．そこでこの組み込みシステムでは，一旦発生した割り込み要求は，少なくとも次に同じ割り込み要求が発生するまでには受け付けられて，対応する割り込み処理が起動されるように設計されていた．これが 6 に示した，割り込み要求が上書きされることがないことという性質の意味である．そのために設計では，各割り込み処理にかかる時間は十分短く，またそれに比べて各割り込み要求の発生間隔は十分長く設定されていた．

ところが，今回はモデルに対する抽象化として，タイマ用変数の変域の縮減などを行ったため，この 6 の性質が損なわれている可能性があった．そこでこの性質を検査項目として設定することで，モデルの妥当性を確保することとした．すなわち，この性質を真とする範囲で，変数の縮約などの抽象化を行った．

本来この性質自体は，モデルの妥当性の検査項目としてではなく，ソフトウェアの動作検査のための項目として設定されるべきである．だが抽象化の必要性から，今回はこの性質の検証を諦めた．そしてこの性質を仮定した範囲でモデルを作成し，ソフトウェアの動作検査を実施することにした．このことについては，9.2 項も参照されたい．

9　ソフトウェアの動作検査

9.1　動作検査

前節に示したようなモデルの妥当性検査を経た上で，本来目的とするソフトウェアの動作検査に入る．

例えば外部信号の正常時，異常時，および，それらの変化時に内部変数が適切な値をとることなどを検査した．

このとき，外部要因モジュールのモデルとして，全く非決定的に変化するものからほぼ周期的に変化するものまで，いくつか用意した．同様に，各種タイマやスイッチによる割り込み要求についても，その発生パターンをいくつか用意した．今回試みた方法を表 4 に示す．

それぞれのパターンには一長一短がある．非決定性の強いものでは，割り込み要求や外部要因の変化がほぼ任意のタイミングで起こり得る状況を模擬できるため，実機で起こり得る状況をほとんどカバーできる．その一方で反例としてあまりに非現実的な挙動が得られる場合があり，その解析は非常に困難である．逆に周期性の強いものでは，反例の解析は比較的容易だが，仕様書に潜在する不備をモデルにおいて喪失してしまう可能性があ

表 5. 検査項目の数と検査結果

検査項目	機能 C	機能 D
用意した数	64	82
検査を実施した数	50	46
結果が真のもの	43	46
結果が偽のもの	4	0
検査不能のもの	3	0

る．また，周期を定めるための変数は，たいていの場合大きな変域をもつ．そのため状態数の増大を招きやすい．

実際のモデルでは，外部要因や割り込み要求ごとに，パターンを使い分けた．また検査項目によっても，いくつかのパターンを使い分けた．

機能 C および機能 D について，動作検査に用いた検査項目の個数と結果を表 5 に示す．

反例解析により検出できた不備の主な原因は，想定外のタイミングでの割り込み発生や変数変化であった．こうした不備は，テストや査読など従来の手法では検出が困難であったと考える．このことは，モデル検査の網羅的な探索を行うという特質が活かされた結果であり，モデル検査の適用が独自の有効性をもつことを示している．

9.2　実時間性に関する性質

また本事例では，システムの実時間性に関する性質の検査を実施するためのモデルを作成したが，検査結果を得ることができなかった．実時間性に関する性質の一例としては，「ある割り込み処理が所定の時間内に終了すること」などがあげられる．こうした性質を検査するためには，数十マイクロ秒で終了する割り込み処理から，数百ミリ秒を周期とするタイマまで，異なる時間スケールを持つ複数のタスクが動くモデルを作らなくてはならなかった．抽象化の方法として，タスクどうしの実行時間の比率に着目し，比率を保ったまま状態数を少くしたモデルを作成した．しかし今回の事例では，タスクの間の時間スケール比は最大で数千になり，モデル検査が可能なサイズのモデルの作成が無理であったため，実時間性の検査は断念した．今後，適切な抽象化技法の開発が課題であろう．

表 6. 検査式の主な形と意味

形	意味
EF p	いずれ p が成立することがある
AG($p \rightarrow$ AF q)	p が成立したら， いずれ必ず q が成立する
AG($p \rightarrow$ A[q U r])	p が成立したら， いずれ必ず r が成立し， それまで q が成立し続ける

9.3　検査式の作成

モデル検査の実施においては，検査項目を時相論理式として記述する必要がある．今回用いた時相論理 CTL は，命題論理に 8 個の時相演算子を加えて拡張したもので，状態変化の必然性や可能性に関する様々な条件を記述することができる．しかしながら，検査したい性質が複雑な場合には，その正確な内容を時相論理式として記述することは難しい．

今回は特にイベントの発生回数に関係する性質で，翻訳が困難なものがいくつかあった．直接翻訳することが難しい性質でも，モデルに検査用の変数をいくつか追加すると，論理式が作り易くなった．その結果，検査したい性質のほとんどは，簡単な構造をした時相論理式で記述できた．特によく用いた検査式の形とその直感的な意味を表 6 に示す．

10　反例解析

10.1　反例解析の手順

反例解析の一般的な手順は以下の通りである．

モデル検査ツールの実行結果が偽となり反例が得られたとき，まずは反例の挙動を解読して，それがシステムのどのような動作を意味しているの

か把握する．

次に，その動作が生じる原因を調べる．そしてその原因を次の 3 つのいずれかに分類する．

(a) モデルの間違い．
(b) 検査項目や検査式の間違い．
(c) システムの間違い．

ここで，(a) モデルの間違いとは，モデルの作り方が不適切であったために，本来は起こりえない挙動が反例として得られたと判断される場合を指す．コード化のミスやモデルの抽象化が不適切であった場合などである．

(b) 検査項目の間違いとは，そもそも設定した検査項目が機能仕様と矛盾したものである場合を指す．検査式の間違いとは，検査項目から時相論理式への翻訳が不適当である場合を指す．

反例の原因が以上のいずれかに該当する場合は，それらの間違いを修正して再びモデル検査ツールを実行する．そうではなく (a) にも (b) にも該当しない場合，(c) に分類する．これはシステムの動作が機能仕様と食い違っているということであり，ソフトウェア仕様書に何らかの不備があるということになる．

10.2　反例解析の困難さ

以上が一般的な反例解析の手順であるが，今回の事例では反例の解析作業に手間取ることが多かった．

まず反例の意味する事象をシステムの動作として把握することは，必ずしも簡単ではない．今回は反例として，数十個の変数の値の変化列で，数百ステップを超えるものがしばしば得られた．これほど大きな反例を一ステップずつ読んで理解するのは，対象のシステムを熟知している設計者が行っても大変な作業であった．今回は反例解析を簡単にするための工夫として，モデルに動作観測用のフラグ変数をいくつか追加し，反例の中のフラグの変化を追跡することで，システムの大まかな動作の理解および不審な個所の探索を行った．しかし，この工夫もシステムについて詳しい知識がないと難しい．

また，反例の原因が前述の 3 つ (a)，(b)，(c) のどれに当てはまるのかを判断するのもしばしば困難な作業であった．

今回得られた反例では，外部要因の極端な変化が原因となっているものがいくつかあった．あまりにも規則的な変化や，あまりにも急激な変化などである．この極端な変化が現実の世界で起こり得るのであれば，(c) と分類され仕様書に不具合があるということになるが，そうでなければ (a) と分類される．こうした外部要因の変化が妥当かどうかの判断は，設計の基本方針にも関わることであり，設計者ですら難しい判断である．

また今回はなかったことだが，仕様書の記述が曖昧で幾通りにも解釈ができる場合には，反例の原因がシステムの不備であるのか検査項目の不備であるのかが不明確となることも考えられる．

もっともこうした判断はモデル検査作業とは独立した，設計現場での作業であるともいえる．モデル検査で得られた反例を解析することにより仕様書に潜在する不具合の可能性を示唆できることは，製品の設計段階において極めて有益であると考える．

11　考察と課題

11.1　組み込みシステムとモデル検査

本事例の対象もそうだが，組み込みシステムは一般にリアクティブシステム [7] である．リアクティブシステムでは，安全性 (safety) や活性 (liveness) が検証すべき重要な性質であり，モデル検査はこうした性質の検証に適している．よって，組み込みシステムに対してモデル検査の適用を試みることは自然である．

このとき，ハードウェアや外部要因といったソフトウェアの動作環境も含めたモデルを作成して検査できることも，組み込みシステムに対するモデル検査適用の利点である．なぜならこれにより試作機の完成を待たずに，ソフトウェアの機能の検査が可能となるからである．またハードウェアや外部要因のモデルを取り替えることで，様々な環境下でのソフトウェアの動作検査が容易にできることもその理由である．

また本事例に見られるようなメイン処理と割り込み処理からなる構成は，組み込みシステムにお

表 7. モデル検査仕様書作成時の作業内容

No.	作業内容
1	機能毎に必要なモジュールを集約する
2	各変数が取り得る状態を抽出して明記する
3	ハードウェアの動作説明を記入する（CPU やタイマなど）
4	割り込みに関する説明を記入する
5	外部要因の特性に関する説明を記入する
6	抽象化可能なところを明示する

いて一般的であろう．個々のモジュールが単体で行う処理は比較的短く単純なものであったとしても，種々の割り込みによってそれらが絡み合った場合，システム全体の挙動を余すことなく捉えることは難しい．このとき，モデル検査の網羅的な状態空間探索を行うという特質が有効である．

本事例ではソフトウェア仕様書に対するモデル検査を，ソフトウェアとその動作環境が一体になったモデルを作成して実施した．その結果，実機試験前の段階で仕様書の不備をいくつか見つけることができた．反例解析により検出できた不備の主な原因は，設計者にとって想定外のタイミングでの割り込み発生や変数変化であった．このことは現場の技術者らにも大きな驚きをもって受け止められた．こうしたことから今回の試行によって，組み込みシステムに対するモデル検査の適用が独自の有効性をもつことを確認できたと考える．

11.2 モデル検査適用のための知見

さらに本事例を通じて，モデル検査を組み込み開発へ適用する際の要領を明らかにし，知見を得ることができた．モデル化および抽象化，検査項目の設定，反例解析に関しては既に述べてきたが，以下にまとめて少し補足する．

モデル検査では，ソフトウェアのモデルに加えて，ハードウェアや外部要因のモデルを作成して，システムと動作環境が一体となったモデルを検査できる．ソフトウェアのモデル化はフロー図にプログラムカウンタを割り付ける手法により，比較的容易にモデル化可能である．むしろ容易でないのは動作環境のモデル化である．ハードウェアのモデルでは，メイン処理と割り込み処理の切り替えや，割り込み禁止区間における割り込み要求の保留などを模擬する必要があり，NuSMV のコードによるいわばプログラミング作業が必要となる．ただこの部分のモデルは，一度作成してしまえば，再利用性は高いと思われる．割り込み要求や外部要因変化の発生については，基本的に非決定的動作を含むモデルを用いればよいが，その加減が難しい．検査の精度，状態数の大きさ，および，反例解析のしやすさを考慮して適切なモデルを作成する必要がある．いくつかのパターンを用意して，複数試してみるとよい．

モデルに対する抽象化は，アドホックな手法であっても，現状のモデル検査ツールの性能からすればやむを得ない．ただし，モデルの妥当性を確保するためにモデルが持つべき最低限の性質を列挙して，これを満たす範囲で抽象化を行うべきである．これを，本事例ではモデルの妥当性検査と呼んだ．本事例において，モデル検査で得られた反例のうちいくつかは，テスト工程における実機試験の結果，実際に起こり得る事象を示していることが確認された．このことは，不備発見のためのモデル検査という観点から，今回のモデル化および抽象化が有効であったことを示すと考える．

また仕様書については，今回作成したモデル検査用仕様書のようなものを最初から用意できればよいだろう．すなわち今回は，モデル検査用仕様書の作成にかなりの時間を費やしたが，最初からモデル検査を前提とした仕様書作りができるならば，この手間が省かれてよい．今回ソフトウェア

仕様書をもとにモデル検査用仕様書を作成したときの作業内容を表7にまとめて示す．

特に，現状のモデル検査ツールの性能を鑑みれば，ソフトウェア仕様書を適切な大きさの機能から構成することが重要である．大きすぎる機能では，モデルの状態数が大きくなりすぎて検査が不可能となる一方，小さすぎる機能では，適切な検査項目が設定できないからである．

ここで，適切な検査項目を設定するためには，機能仕様書とソフトウェア仕様書の間の対応関係が明らかでなければならない．

機能仕様書の内容がソフトウェア仕様書に記載されたアルゴリズムによって実現されているかどうかを確かめるとき，システムのモデル化はソフトウェア仕様書に基づいて行う一方で，検査項目の選定は機能仕様書に基づいて行う必要がある．だが，機能仕様書では自然言語による機能説明がなされ，ソフトウェア仕様書ではフロー図などの実装に近い言葉により動作が記述されるのが通常である．そのため，機能仕様書から選び出した性質をモデル検査用の検査項目とするためには，その性質をソフトウェア仕様書で用いられる変数の値の変化条件として書き直す必要が生じる．

今回は，機能仕様書とソフトウェア仕様書の間のこうした対応関係が必ずしも明確でなかったために，モデル検査用仕様書の作成にはかなりの労力がかかった．そこで設計当初から，機能実現のためにプログラムの変数が果たす意味や役割を明確にするなど，両者の対応関係を明らかにしておくとよいだろう．

11.3 今後の方策

今後，モデル検査を本格的に実用化させるには，さらなる理論の整備と普及に向けた方策が必要である．

理論の整備としては，より強力な抽象化技法や実時間性検査のためのモデル検査アルゴリズムの開発が期待される．また，システムの部分的な検査により全体の性質を保証するための理論が発展すれば，抽象化が必要なくなるかもしれない．新たな理論に基づくより高性能なモデル検査ツールの登場も期待される．

モデル検査を普及させる方策としては，モデル検査の適用に向く仕様書の構成の検討や，技術者向けの教材開発，セミナーの実施などが挙げられる．

こうしたことに加えて，さらに事例研究を積み重ねていくことも極めて重要であろう．そもそもモデル検査がいくら理論的に発展しようとも，それがソフトウェアの開発現場で生じるどのような問題に対しどのように有効であるかということは，決して自明ではない．今回は，モデル検査をソフトウェア仕様書の検証に利用し，想定外の割り込みタイミングなどに起因する不備が検出可能であるという事例が得られた一方で，実時間性の検査には課題が多いことが分かった．このような結果を蓄積することは，真の実用化に向けて欠かせない取り組みであろう．また，今回得られた知見を一定の方法論にまでまとめるためにも，様々な事例を通じてその適用可能性を検討する必要がある．

事例研究の遂行そのものにも多くの課題がある．モデル検査の長所や短所を他の検証手法と比較することは興味深いが，何をどのように比較したらよいであろうか．組み込み開発におけるモデル検査の事例はまだ少ないし，他の手法と比較しようにも適当な基準が見当たらない．また，実用化のための事例研究となると，モデル検査の有効性を費用対効果により分析することが不可欠であろうが，いかに実施すればよいだろうか．また，モデル検査を他の形式手法やモデルベース開発と組み合わせて実施することは有効であると考えるが，いかにして実施したらよいであろうか．

こうした課題は実用化を視野に入れた事例研究が独自に抱えるものであり，今後の取り組みの中で解決策を探らねばならない．

謝辞

本稿の内容は，部分的に文部科学省科学技術振興調整費 (若手任期付研究員支援) の支援を受けた研究成果である．また，本研究の一部は企業との共同研究により遂行された．記して感謝申し上げる．

参考文献

[1] Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, Ph. and Mckenzie, P.: *Systems and Software Verification: Model-Checking Techniques and Tools*, Springer, 2001.

[2] Cimatti, A., Clarke, E. M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R. and Tacchella, A.: NuSMV2: an Open Source Tool for Symbolic Model Checking, in *Proc. International Conference on Computer-Aided Verification* (*CAV 2002*), Copenhagen, Denmark, July, 2002, pp. 27-31.

[3] Clarke, E. M., Grumberg, O. and Peled, D. A.: *Model Checking*, The MIT Press, 1999.

[4] Holzmann, G. J.: *The Spin model checker: Primer and Reference Manual*, Addison-Wesley, 2003.

[5] Huth, M. and Ryan, M.: *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge University Press, 2000.

[6] 木下佳樹: 産業技術とシステム検証, 産業技術総合研究所算譜科学グループ研究速報, AIST-PS-2003-02, February, 2003.

[7] 大崎人士, 木下佳樹, 高井利憲, 高橋孝一, 古澤仁: リアクティブ・システムの検証法, 産業技術総合研究所算譜科学グループ研究速報, AIST-PS-2001-04, November, 2001.

[8] 篠崎孝一, 水口大知, 石井健志: 組込みソフトウェア開発のイン-デザイン モデル検査, 産業技術総合研究所算譜科学グループ研究速報, AIST-PS-2004-01, January, 2004.

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
(算譜科学研究速報)
発行日：2005 年 1 月 20 日
編集・発行：独立行政法人産業技術総合研究所関西センター尼崎事業所
システム検証研究センター
同連絡先：〒661-0974 兵庫県尼崎市若王寺 3-11-46
e-mail：informatics-inquiry@m.aist.go.jp
本掲載記事の無断転載を禁じます

A Case Study of Applying Model Checking to Embedded Software Development
(in Japanese)
(Programming Science Technical Report)
January 20, 2005
Research Center for Verification and Semantics (CVS)
AIST Kansai, Amagasaki Site
National Institute of Advanced Industrial Science and Technology (AIST)
3-11-46 Nakouji, Amagasaki, Hyogo, 661-0974, Japan
e-mail:informatics-inquiry@m.aist.go.jp
・Reproduction in whole or in part without written permission is prohibited.