社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－ＳＥ－160 (13)
2008－EMB－ 9 (13)
2008／6／20

モデル検査技術を活用したソフトウェア設計・検証手法に関する考察

岸知二† 高橋弘†† 徳田寛和††

モデル検査技術のソフトウェア検証への適用が活発に検討されている。本稿ではモデル検査技術を用いたソフトウェア検証の試行について報告するとともに、検証を想定した設計の在り方などについて検討する。

On Software Design and Verification Methods utilizing Model Checking Techniques

TOMOJI KISHI,† HIROSHI TAKAHASHI†† and HIROKAZU TOKUDA††

Model checking techniques are considered to be promising techniques for software verification. In this paper, we introduce a trial of software verification utilizing model checking techniques, and examine software design that facilitates the software verification.

1. はじめに

近年ソフトウェアの品質、特に信頼性や安全性に対するメーカやユーザの関心が高まっているが、そうした背景の元、ソフトウェア検証への形式手法、特にモデル検査技術[1]の適用への取り組みが、産学において活発化している[2),4)]。組込みソフトウェアは一般に複雑な動的ふるまいを持つことが多いため、そのふるまいの検証にモデル検査技術の適用が有用ではないかとの期待がもたれている[5)]。

我々は組込みソフトウェアの設計が意図したとおりのふるまいをするかどうかをモデル検査技術を用いて検証する試みを行った。このソフトウェアは本来モデル検査技術による検証を意図して作られたものではなかったため、検証の過程でいくつかの問題が生じた。この経験を踏まえ、モデル検査技術を適用することを想定した設計の在り方や、それに関わる課題などについて検討した。本稿では、その経験と、検証を想定した設計に関する考察について報告する。

2章では、どのような検証を行ったかを説明する。3章では、検証の結果について報告する。4章では、本検証の試行結果に基づき、検証を想定した設計に関して検討する。5章では、検証を行う際に考慮すべき課題について議論する。

† 北陸先端科学技術大学院大学大学 情報科学研究科
School of Information Science, Japan Advanced Institute of Science and Technology
†† 富士電機アドバンストテクノロジー (株)
Fuji Electric Advanced Technology Co.,LTD.

2. 状態遷移に関する設計検証

本章では、我々が試行したモデル検査技術によるソフトウェア検証の内容について説明する。

2.1 検証の概要

検証の概要は以下のとおりである。

2.1.1 対象ソフトウェア

機器の状況を複数のセンサーでとらえ、状況に応じて適切な処理を行うソフトウェアである。ソフトウェアの内部では機器の状況の変化を状態遷移モデルとして捉え、その状態変化に応じて適切な処理を行う。状態は複数の変数値の組み合わせで表現されている。状態遷移を引き起こすイベントは複数センサーからの入力に基づいて定義される。例えばある入力は二値を持ち、これが一方の値になることがイベントとして定義される。またある入力はアナログ値を持ち、これが一定の値を超えることがイベントとして定義される。

なお検証にはCコードレベルに近い詳細な設計情報を利用した。

2.1.2 設計情報

設計意図としてのふるまいは状態遷移モデルとして与えられた。状態遷移モデルは状態とイベントに基づいて定義されるが、状態がどのような変数値の組み合わせで表現されているか、イベントと入力センサーの値との対応関係は示されていなかった。これらの情報

は、技術者に対するヒアリングの形で補足した。

2.1.3 検証の目的と方針

検証の目的は、対象ソフトウェアの設計が状態遷移モデルとして与えられる仕様どおりのふるまいを示すかどうかを検証することである。これをモデル検査技術を利用した以下の二通りの検証を組み合わせることで検証した。検証には SPIN[3] を用いた。

- 検証 1:意図した状態遷移が実現されているかどうかの検証。想定する状態遷移を起こし、ソフトウェア内部の変数がその状態を示しているかどうかを確認する。
- 検証 2:意図していない状態遷移が起きないことの検証。起こりうる環境変化（センサ値の変化）を起こし、ソフトウェア内部の状態変化を監視することで意図した遷移のみ が起こっているかどうか確認する。

以下、その詳細について説明する。

2.2 検 証 1

検証 1 では、意図した状態遷移が実現されているかどうかを、以下の手順で確認する。

(1) 設計対象のモデル化：対象ソフトウェアの設計を Promela で表現する。本稿ではこれを対象モデルと呼ぶ。C 言語での実装に近い詳細レベルで与えられた設計情報に基づき、変数や入力値の判断などを同等の Promela で実現した。

(2) アクタの作成：センサ値を変化させるアクタを Promela で実現する。このアクタは意図した状態遷移に基づき、起こりうる遷移を非決定的に起こす。

(3) assert 文の作成：アクタは状態遷移を引き起した個所に、対象モデル中の変数値群が遷移先の状態を表しているかどうかを確認する assert 文を挿入し、正しい状態に遷移しているかどうかを確認する。

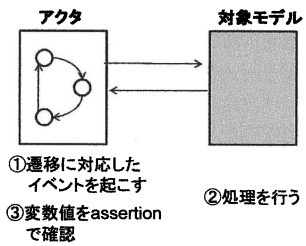

図 1 検証 1 の概要

図 1 に検証 1 の概要を示す。アクタは遷移に対応したイベントを対象モデルに送り、対象モデルはそのイベントに応じた処理を行い、アクタは変数値を参照して意図した状態に変化しているかどうかを assert 文で確認する。

2.3 検 証 2

検証 2 では、意図していない状態遷移が起きないかどうかを、以下の手順で確認する。

(1) 設計対象のモデル化：これは検証 1 と同様である。

(2) アクタの作成：センサ値を変化させるアクタを Promela で実現する。このアクタは起こりうるセンサ値の変化を非決定的に起こす。ただしアナログ的な値を持つ入力に対しては、イベントの定義に照らして値を抽象化する。例えばある値が一定の値以上になることをイベント発生と捉える可能性がある場合には、値をその値より大きい、値と同じ、値より小さいというように抽象化し、その 3 種類の値を非決定的に起こした。

(3) モニタの作成：対象モデル中の変数値群を監視してどの状態にあるかを監視する。遷移を監視するために、常に前回の状態と今回の状態の二つを保持する。

(4) assert 文の作成：現状態判断の処理の後に、前状態と現状態の間に遷移が定義されているかどうかを確認する assert 文を挿入し、正しい遷移のみが起こっているかどうかを確認する。

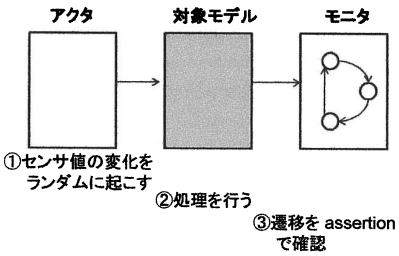

図 2 検証 2 の概要

図 2 に検証 2 の概要を示す。アクタはセンサ値の変化を起こし、対象モデルはその時のセンサ値に基づいた処理を行い、モニタは変数値を参照して意図した状態遷移のみがおこっているかどうかを assertion で確認する。

2.4 例 題

簡単な例題で本検証の方法を示す。この例題は実際に検証を試行したシステムそのものではないが、本稿での説明のために、今回試行した方法の本質的な部分

のみを抜き出し、簡略化して作成したものである。

このシステムはひとつの入力を持ち、そこからはセンサ値 (v) を得ている (図 3)。

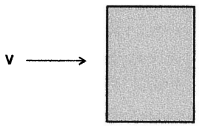

図 3 例題システム

図 4 はシステムの持つ状態遷移である。

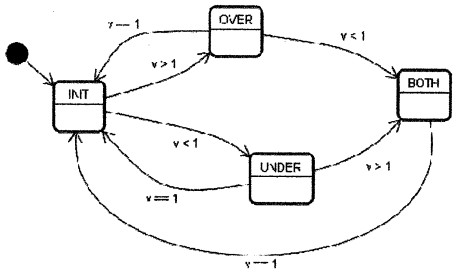

図 4 例題システムの状態遷移モデル

本システムは二つのブール変数 c0, c1 を利用して状態を管理するように設計されている。入力値はアナログ値であるが、特定の値 (ここでは 1) との比較でイベントが定義されているため、入力値は 0(特定の値より低い)、1(同じ)、2(高い) の 3 値に抽象化されている。表 1 に状態と変数値との対応を示す。

表 1 変数値と状態の対応

状態	c0	c1
INIT	false	false
OVER	true	false
UNDER	false	true
BOTH	true	true

図 5 は本システムの対象モデルである。なお provided 文は対象モデルとアクタとを交互に実行するために利用している。変数 turn が PTURN であるときに対象モデルが実行され、その時点での入力値に基づいて必要な処理を行い、実行権を放棄する (ここでは ETURN としてアクタに実行権を移しているが、順序は検証 1 と検証 2 とで異なる)。

検証 1 では、図 4 で示す状態遷移モデルどおりの遷移を起こすようにの入力値を変化させるアクタを作る。図 6 は、アクタの例である。ここではその時点での状態で定義されている遷移に対応する入力値を設定

```
byte v0 = 1 ;
bool c0 = false ;
bool c1 = false ;
bool flag = false ;

active proctype P()   provided(turn==PTURN)  {
L0:
            if
            :: (c0==false && v0 > 1) -> c0 = true
            :: (c1==false && v0 < 1) -> c1 = true
            :: (v0==1) -> c0=false; c1=false
            :: else -> skip
            fi ;
            turn = ETURN ;
            goto L0;
}
```

図 5 対象モデル

```
active proctype E() provided(turn==ETURN) {
INIT:       assert(c0==false && c1==false) ;
            if
            :: true -> v0=2; turn = PTURN ; goto OVER
            :: true -> v0=0; turn = PTURN ; goto UNDER
            fi ;
OVER:       assert(c0==true && c1==false) ;
            if
            :: true -> v0=1; turn = PTURN ; flag = false; goto INIT
            :: true -> v0=0; turn = PTURN ; flag = true; goto BOTH
            fi ;
UNDER:      assert(c0==false && c1==true) ;
            if
            :: true -> v0=1; turn = PTURN ; flag = false; goto INIT
            :: true -> v0=2; turn = PTURN ; flag = true; goto BOTH
            fi ;
BOTH:       assert(c0==true && c1==true) ;
            if
            :: true -> v0=1; turn = PTURN ; flag = false; goto INIT
            fi ;
}
```

図 6 検証 1 でのアクタ

```
active proctype E() provided(turn==ETURN) {

L0:
            if
            :: true -> v0 = 0
            :: true -> v0 = 1
            :: true -> v0 = 2
            fi ;

            turn = PTURN ;
            goto L0;
}
```

図 7 検証 2 でのアクタ

し、実行権を対象モデルに引き渡す。

検証 2 でのアクタは状態に関わらず起こりうる入力値の変化をランダムに起こす (図 7)

図 8 はモニタである。ここでは変数値 (c0, c1) をモニタし現状態を判断するとともに、保存した前状態から現状態への遷移が許されているものかどうかを assert 文で確認している。なお、検証 2 では変数 turn

を使い、アクタ、対象モデル、モニタの順に実行を制御する。

```
#define SINIT 0
#define SOVER 1
#define SUNDER 2
#define SBOTH 3
byte cstate = SINIT;  byte pstate = SINIT

active proctype M() provided(turn==MTURN) {
L0:         pstate = cstate ;
            if
            :: (c0== false && c1==false) -> cstate = SINIT
            :: (c0== true  && c1==false) -> cstate = SOVER
            :: (c0== false && c1==true) -> cstate = SUNDER
            :: (c0== true  && c1==true) -> cstate = SBOTH
            fi ;
            assert(pstate==SINIT && cstate==SINIT
                  || pstate==SINIT && cstate==SOVER
                  || pstate==SINIT && cstate==SUNDER
                  || pstate==SOVER && cstate==SINIT
                  || pstate==SOVER && cstate==SBOTH
                  || pstate==SUNDER && cstate==SINIT
                  || pstate==SUNDER && cstate==SBOTH
                  || pstate==SBOTH && cstate==SINIT
                  || .... <<省略>>
            );
            turn = ETURN ;
            goto L0
}
```

図 8 検証 2 でのモニタ

3. 検証結果

前章で述べた検証を行った結果を以下に示す。

3.1 検証 1 の結果

検証モデルの作成にあたっては、設計情報から Promela への変換ルールを決めて手続き的に行い、極力アドホックな変換を避けるようにした。なお対象となるソフトウェア設計では、センサからの入力値を確定するために一定時間入力値が同じ値であることを監視して実際のセンサ値を確定するなど、入力の確定に関わる処理部分があったが、今回の試行ではこの部分は抽象化し、入力値は確定的に変化するものとしてモデル化した。この部分をモデル化に含めると状態数が大きく増加する危険性があるからである。

検証では当初 assertion 違反が検出された。これはイベントとセンサ入力との対応づけ、状態と変数値との対応づけに関する設計情報に不正確だった部分があったためである。インタビューの過程でその情報を補正することにより基本的には検証を進めることができた。対象ソフトウェア中では多くの変数が使われているが、状態と変数値の対応づけにおいて、当初関係のないと思った変数が条件に関わっているなどの状況があり、その確認に時間がかかった。

3.2 検証 2 の結果

検証では assertion 違反が検出された。すなわち意図していない遷移が発生した。この原因のひとつは、複数のセンサ値の入力を相互独立に非決定的に起こしたため、現実にはありえないセンサ値入力の組み合わせが発生したためである。ありえない入力値の組み合わせに対しては、それをソフトウェア内部では考慮しないこともあり得るため、そうしたありえない入力値群が入った場合に意図しない遷移が発生したと考えられる。もちろんソフトウェアの設計に間違いがあったために意図しない遷移が発生したことも当然原因として考えられる（その可能性があるからこそ検証をしている）。

今回の試行では、状態遷移モデル上で明示的に定義されたイベントの系列に対応するセンサ値群の変化に対しては意図通りの状態変化が起こっていることが検証 1 で確認されているので、今回の assertion 違反の原因は明示化されていないセンサ値群の変化に関わるものである。明示されていない理由としては、そのような変化があり得ないことが暗黙の了解になっているから、システムの動作条件として考慮すべき変化の範囲が暗黙裡に設定されているから、などといくつかの理由が考えられる。設計が本来想定している変化のパターンの範囲がどこにあるのか、その範囲の識別が行えないと、本検証は困難であることがわかった。

4. 検証を想定した設計に関する考察

前章で紹介した検証の試行に基づき、検証を想定した設計に関し、仕様定義とフトウェア構造の観点から考察する。

4.1 仕様定義

4.1.1 状態遷移の定義

今回の検証ではシステムのふるまい仕様は状態遷移モデルとして与えられた。状態遷移モデルで与えられる仕様とのつきあわせは、モデル検査技術を用いた検証においては典型的なものであるが、状態遷移モデルを定義するためにはシステムの持つ状態と、状態間の遷移を引き起こすイベントを定義しなければならない。例えば「正常」という状態を定義するならば、その状態がシステムのどのような状況を表すのかを明確に定義しなければならない。同様に「異常入力」というイベントを定義するならば、それがシステムに対するどのような事象を示すのかを明確に定義しなければならない。

モデル検査技術を使った検証においては典型的なふるまいだけでなく、起こりうるイベントと状態との組

み合わせに関する網羅的なふるまいを定義する必要があるが、特に本事例のようにイベントがアナログ値に基づいて定義される際には、その値に対してどのような判断が付随するかを整理する必要があり、その作業は煩雑である。例えばある状態では入力値が 10 を超えることがイベントとして捉えられ、別の状態ではその値が 20 を超えることがイベントとして捉えられるというように、イベントの識別が複雑な状況もある。あるいはシステムによってはそうした閾値自体が最終的なチューニングの段階で決定されることもある。もちろんそうした状況を考慮した漏れの少ない設計を工夫することは可能であるが、状態遷移設計に不慣れな分野の技術者に対しては作業の指針がなければ間違いを持ちこみやすくなる。

以上より、もっとも基本的なことではあるが、対象システムを状態遷移モデルとしてモデル化するにあたり、状態やイベントの明確な定義と、網羅性をもった定義が必要であることがわかる。

4.1.2 外部制約の定義

検証 2 では検証におけるアクタからの入力値群がありえない値の組み合わせを含んでいたために、意図しない状態遷移が起こり assertion 違反が発生した。こうしたありえない値の組み合わせなど、外部制約を定義しないと厳密な検証が困難な場合がある。

こうした外部制約は、外界の性質から起因するものと、システム仕様の設定からくるものとが考えられる。外界の性質に起因するものとは、たとえば電源が入っていない装置からは信号がやってこないとか、排他的な状況を捉える二つのセンサの値は同時には ON にならないなどといった制約である。こうした制約の中には自明なものも含まれるが、システム外界の物理的な環境の性質に依存するものなど、明確に定義することが困難なものもある。その場合は広めの想定をするなど妥当な制約を設定することが必要となる。

一方システム仕様の設定からくるものとは、たとえばあるセンサが故障して信号が入らなくなる状況もあるが、そういう状況は別の手段で検知するため、対象ソフトウェアにおいてはそうした状況は考えなくてよい、といった制約である。これらは開発者の仕様設定の意図であるため、外部制約よりはより明示的に定義しうるものではあるが、得てして暗黙裡に決められていることなども多く、そうした制約の明示化の作業が必要となる。

4.2 ソフトウェア構造

4.2.1 状態遷移の実現手法

検証を行うためには、意図した状態遷移モデルが、対象ソフトウェアの設計にどのように対応で付けられ実現されているかその設計方針が明確になっていることが必要である。

状態遷移モデルをソフトウェアとして実装するプログラミングレベルでの技法としては、例えば状態変数を持たせる方法、プログラムの分岐 (switch 文等) で実現する方法などがある。状態モデルとの対応を明確にするために、状態モデルの情報をテーブルとして一元管理する、それを抽象化して次状態を決定する関数を用意する、あるいは状態パターンを用いるなど、多様な方法があるが、いずれの方法であれ、一貫した方針で状態遷移が実現されているならば（モデル検査の方法や利用ツールの事情に応じた相性はあるにしても)、それを参照しながら検証することが可能となる。逆にその方針に一貫性を欠けばそれは困難となる。

4.2.2 状態遷移実現部分の分離

実際のシステムでは、入力値の組み合わせからイベントを判定する処理やセンサ入力の確定を待つ処理など、外部とのインタフェースに関わる処理なども含まれている。こうした部分と状態遷移実現部分を明確に分離する設計にすることで、状態モデルの実現が容易になるだけでなく、検証が容易になり不要な状態数増加を抑えることができる。

またエラー数が一定数を超えたら状態を遷移するような処理の場合、そのエラー数をカウントするカウンタを状態に含めると状態数が増加する。またカウント数をいくつにするかということ自体がチューニングの対象となることもある。こういう場合には状態遷移上は「一定数を超える」という抽象的なイベントを定義し、具体的なカウント数を状態遷移モデルとは分離することが望ましい。

このように状態遷移の実現部分をモジュールとして分離するだけでなく、適切な抽象化を施すことでより検証がやりやすくなると考えられる。

5. 議　　論

今回の検証試行においては、対象システムの仕様は状態遷移モデルとして与えられた。状態遷移モデル中ではイベントに対する状態遷移が定義されるが、イベントは対象システムの持つ複数の入力に基づいて定義されている。

図 9 は、これらの関係を模式的に示したものである。検証対象は入力 I を受け取り、また何らかの内部状態 V を持つ。入力は一般に複数あり得、それらは離散的な値をとることもあれば連続的な値をとることもある。内部状態は内部に保持する変数の値であった

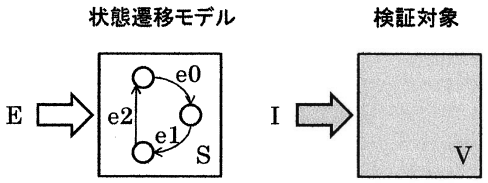

図 9 状態遷移モデルと検証対象

り、ソフトウェアの実行位置であったりする。一方状態遷移モデルは検証対象を、イベント E と状態 S を用いてモデル化する。

検証対象が状態遷移モデルと同様のふるまいをするかどうかを検証するためには、この異なったふたつの世界を対応づけなければならない。すなわち検証対象に対する入力 I と状態遷移モデルにおけるイベント E、検証対象の内部状態 V と状態遷移モデルにおける状態 S との対応づけが必要である。この対応づけがどのようになっているかによって、検証する内容や方法に違いが出てくる。

検証対象に対する入力 I と状態遷移モデルにおけるイベント E の対応について考えてみる。例えば検証対象はふたつの入力 I_0 と I_1 を持つとし、それぞれ入力値 0, 1 という離散的な値をとるとする。<I_0, I_1> の値の組を考え、<0,0> を e0, <0,1> を e1, <1,1> を e2 と対応づけたとする。また状態遷移モデルモデル上で明示的に示されているイベントの系列は e0 → e1 → e2 →…だとする。

様々なイベント系列に対して網羅的な検証を行う際に、そのイベント系列の与え方としては以下が考えられる。

- 系列 0：状態遷移モデル上で明示的に示されているイベント系列を与える。上記では e0 → e1 → e2 →…を与える。
- 系列 1：E 中に含まれるイベントから生成されるイベント系列を与える。上記では e0, e1, e2 から生成されるイベント系列を与える。
- 系列 2：I 中に含まれる入力値から生成されるイベント系列。上記では I_0、I_1 それぞれが 0 と 1 をとることから、その組み合わせから生成されるイベント系列を与える。

検証 1 では、系列 0 を用いて検証を行った。すなわち明示的に示されているイベント系列に対しての検証を行った。これで検証できることは、明示的に定義されていることが、検証対象でもなりたっているかどうかということである。

検証 2 では、系列 2 を用いて検証を行った。すなわち起こりうる入力系列に関する検証を行った。これにより (1) 系列 1 に含まれ系列 0 には含まれないイベント系列に関する検証 (e0 → e2 → e0 →…等)、(2) 系列 2 に含まれ系列 1 に含まれないイベント系列に関する検証 (<1,0> を含む系列等) に関する検証が行われることになる。この (1) および (2) がまったく存在しない（すなわち系列 0 が定義されているイベントのすべての系列を含み、かつありえる入力値の組み合わせがもれなくイベントに対応づけられている）状況もありえるが、現実には大きなソフトウェアでは、その差分が存在することも多い。

今回検証 2 がうまくいかなかった理由を上記の観点から考えると、(1) および (2) は明示的に意図として定義されていないのであるから、そうした状況で何が起こるか未定義であったからといえる。今回は、状態遷移モデルに書かれていないことは起こらないという仮定を検証したが、その仮定が正しいかどうかは、検証で確認できることではなく、その前提として意図を明確にすべきことだったといえる。

6. おわりに

本稿では、ソフトウェアのふるまいをモデル検査技術を用いて検証した試行の経験と、検証を想定した設計に関する考察について報告した。モデル検査においては、明示的に何を認識しているのか、対象をどのように抽象化しているのか、などを注意深く検討することが必須であり、そうした検討をせずにいたずらに検証を行うことは意義を損なうことがわかった。今後はこうした検討をわかりやすく行うための支援について考えたい。

参考文献

1) Clarke, E., Grumberg, O., Peled, D.: Model Checking: MIT (1999).
2) 第四回システム検証の科学技術シンポジウム予稿集, 日本ソフトウェア科学会ディペンダブルシステム研究会, (2007).
3) Holzmann, G.J.: The SPIN Model Checker - Primer and Reference Manual, Addison-Wesley (2004).
4) 塚田恭章編: フォーマルメソッドの新潮流, 情報処理, Vol.49, No.5, (2008).
5) 岸知二, 金井勇人:組込みソフトウェア設計検証へのモデル検査技術の適用と考察, IPA/SEC, SEC Journal 12 号, (2007).