情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2009-EMB-14 No.4
2009/7/24

モデル検査による設計検証と整合テスト

青 木 利 晃[†1]　NGUYEN Tam Thi Minh[†2]

モデル検査は，扱える状態数の限界などから，実装よりは状態数が少い設計モデルの検証に適していると言われている．一方で，検証した設計モデルに基づいて実装する際，設計検証で保証した性質は，実装後も成立しているべきである．そこで，我々は，設計モデルと実装が整合していることを保証する手法について研究を行っている．ここでの整合性とは，設計モデルにおいてモデル検査により保証した性質が，実装においても成立することである．この整合性が成立することを，従来から研究されている整合テスト，特に，Automata-Theoretic Conformance Test の枠組を拡張してテストすることを試みる．

Study on Model Checking Design Models and Conformance Testing

Toshiaki Aoki[†1] and NGUYEN Tam Thi Minh[†2]

It is known that model checking is suitable for verifying design models because the number of states of the design models have is much smaller than that of their implementations. In this style of the verification, properties ensured by verifying the design models should hold in the implementations. Thus, we are studying on ensuring conformance between the design models and implementations. In this study, the conformance means that the properties verified in the design models by the model checking hold in the implementations. To ensure the conformance, we extend conformance test, especially, Automata-Theoretic Conformance Test which has been studied for a long time.

†1 北陸先端科学技術大学院大学
Japan Advanced Institute of Science and Technology
†2 北陸先端科学技術大学院大学 (昨年度まで)
Japan Advanced Institute of Science and Technology(until last academic year)

1. はじめに

ソフトウェア開発では，一般に,「要求分析」,「設計」,「実装」といった工程に分けられる．それぞれの工程では，単に，ドキュメントやコードを作成するだけでなく，それらの品質を保証するのが望ましい．従来の開発手法では，ソフトウェア実装後に検証作業が集中していたが，上流工程で決定するソフトウェアの構造に関する性質は，実装前に検証すべきである．そこで，これまでに，設計工程で作成する設計モデルをモデル検査を用いて検証する手法について研究を行ってきた．しかしながら，検証した設計モデルに基づいて実装する際，設計検証で保証した性質は，実装後も成立していなければならない．よって，検証した結果をソフトウェア実装後も保証する仕組みが必要である．このような仕組みには２つのものが考えられる．１つ目は，**MDA**(*Model Driven Architecture*) を用いる手法である．この手法では，検証した設計モデルに基づいて，保証した性質を崩さないように詳細化したり，複数の設計モデルを合成し，実装と同型な実装モデルを作成する．そして，作成した実装モデルからソースコートを自動生成するのである．しかしながら，検証した性質を崩さないように詳細化や合成するのは困難である．また，ほぼ設計モデルと同型な実装モデルを作成すると，パフォーマンスが低下し，場合によっては，実装対象プラットフォームの制約を満たせ無いなどの状況が生じる恐れがある．そのため，本研究では，２つ目のアプローチをとる．この手法では，設計モデルに基づいて，人手で最適化やプラットフォームへの適合を行い，その後，設計モデルと整合してるかどうか，回帰的な検査を行う．回帰的な検査には**整合テスト** (*Conformance Test*) を用いる．従来からの整合テストでは，ブラックボックステストを前提としている．しかしながら，この前提では，完全な整合性，もしくは，精度の高い整合性を保証するのは困難である．そこで，本研究では，実装の内部を一部参照する，グレーボックステストを前提とした，整合テスト手法を提案する．

2. Automata-Theoretic Conformance Test

整合テスト手法とは，仕様と実装が整合していることをテストする手法の総称である．また，整合テスト手法の１つとして，*Automata-Theoretic Conformance Test* (**オートマトン理論に基づいた整合テスト**)[1)] がある．ここで，２つの入出力付きオートマトン A と B を考える．A は仕様を表現するオートマトンであり，そのすべての情報は参照することができる．一方，B は実装を表現するものであり，ブラックボックスである，すなわち，内部状態や遷移は見ることができないが，入力を与えた時の出力は観測できるものとする．これらの前提

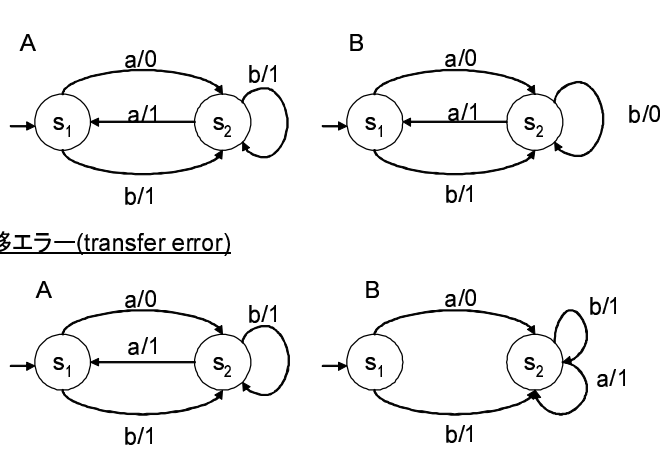

図 1 操作エラーと遷移エラー

で，B が A と同じであることを保証するためのテストを考えるのが，Automata-Theoretic Conformance Test の問題である．しかしながら，これら以外になにも仮定無しでは，この問題は解決できない，つまり，決定不能問題である．そこで，以下の仮定を置くのが一般的である．

(1) A は強連結である．
(2) A は最小 (minimal) である．
(3) B は A と同じ入力記号を持つ．
(4) B はテスト中に変更されない．
(5) B は A より多くの状態を持たない．

5 番目の仮定は，際限無く多くの状態を想定しなければならないことを避けるためのものである．もし，B が A より多くの状態を持つのであれば，その余分な状態 (extra state) に到達するようなテストケースを作らなければならない．もし，余分な状態の数がわからなければ，どれくらい多く実装を遷移させればすべての状態に到達するかわからないので，決定不能問題となってしまう．一方，B は A より多くの状態を持たない場合は，考えられる誤りとしては**遷移エラー** (*transfer error*) と **操作エラー** (*operation error*) のみであり，これらが無いことが保証できれば，A と B が同値であることが保証できるのである．ここで，操作エラーとは，遷移において誤った記号を出力することであり，遷移エラーとは，誤った次状態に遷移することである．図 1 に，それぞれのエラーの例を示す．上の 2 つの状態モデルでは，A では状態 s_2 における自己遷移において 1 を出力するのに対し，B では 0 を出力している．よって，これは遷移エラーである．下の図では，A では状態 s_2 から状態 s_1 に入力 a，出力 1 で遷移しているのに対して，B では状態 s_2 から状態 s_2 への遷移になっている．よって，これは遷移エラーである．また，仮定 5 より少し弱い，余分な状態数の上限がわかる，という仮定を置く場合もある[2)]．

3. 設計と実装の整合性の定式化

3.1 設計と実装

この Automata-Theoretic Conformance Test を，設計モデルと実装の整合性保証に応用することを考える．本研究では，設計モデルは，モデル検査により望ましい性質が成立することが保証されているものとし，その性質の集合を P とする．ここでは，線形時間論理により性質が書かれているものとする．そして，設計と実装の整合性とは，従来からの入力に対して期待する出力が得られることと，P のそれぞれの性質が実装においても成立することである．本研究では，特に，後者に注目する．

従来からの整合テストの枠組と同様，設計と実装は，入出力付きオートマトンで表現する．一方で，モデル検査を用いて設計モデルの検証を行う場合，P のそれぞれの性質が成立するかどうかは，状態で成立する**原始命題** (*atomic proposition*) に依存している．そこで，クリプケ構造と同様に，入出力付きオートマトンの状態に原始命題集合を割り当てるラベル関数を導入する．

定義 3.1 設計と実装のオートマトン
設計モデル DM を以下のように表現する．ただし，S_D は設計の状態の集合，I は入力記号の集合，O は出力記号の集合，A は原始命題の集合である．

$$DM = (S_D, I, O, \delta_D, \lambda_D, s_{D0}, L_D)$$
where
$$\delta_D : S_D \times I \to S_D,$$
$$\lambda_D : S_D \times I \to O,$$
$$L_D : S_D \to 2^A, s_{D0} \in S_D$$

実装 IM を以下のように表現する．ただし，S_I は実装の状態の集合，A は原始命題の集合

である．

$$IM = (S_I, I, O, \delta_I, \lambda_I, s_{I0}, L_I)$$
where
$\delta_I : S_I \times I \to S_I$,
$\lambda_I : S_I \times I \to O$,
$L_I : S_I \to 2^A, s_{I0} \in S_I$

3.2 環境の取り扱い

整合テストでは，対象はオープンシステム (open system)，すなわち，入力を与えると動作するというものに対してテストを行う．一方，設計モデルのモデル検査は，通常，環境と呼ばれる，対象への入出力の部分を作成して，閉じられたシステム (closed system) にして検査を行う．そのため，ここでの入出力記号は，環境と対象の間でやりとりされるものである．

定義 3.2 環境のオートマトン
環境 E を以下のように表現する．ただし，S_E は環境を構成する状態の集合である．

$$E = (S_E, I, O, T, s_{E0})$$
where
$T \subseteq S_E \times I \times S_E \times O, s_{E0} \in S_E$

以上の定義からわかるように，設計モデル DM と実装 IM はすべての遷移が決定的であるのに対して，環境は遷移が非決定的でも良い．

定義 3.3 環境と設計モデル，環境と実装の合成
設計モデルを環境を用いて closed system にした状態遷移モデル $DM||E$ を以下により表現する．

$$E||DM = (S_D, I, O, \delta_{E||D}, \lambda_{E||D}, s_{(E||D)0}, L_{E||D})$$
where
$\delta_{E||D} : S_D \times I \to S_D$,
$\lambda_{E||D} : S_D \times I \to O$,
$L_{E||D} : S_D \to 2^A, s_{(E||D)0} \in S_D$

ただし，合成した状態は $S_E \times S_D$ の要素で表現されるが，ここでは，それらを S_D に射影している．
同様に，実装を環境を用いて closed system にした状態遷移モデル $E||IM$ を以下により表現する．

$$E||IM = (S_I, I, O, \delta_{E||I}, \lambda_{E||I}, s_{(E||I)0}, L_{E||I})$$
where
$\delta_{E||I} : S_I \times I \to S_D$,
$\lambda_{E||I} : S_I \times I \to O$,
$L_{E||I} : S_I \to 2^A, s_{(E||I)0} \in S_I$

ただし，合成した状態は $S_E \times S_I$ の要素で表現されるが，ここでは，それらを S_I に射影する．本来は，合成状態の作り方を形式的に定義する必要があるが，その定義は一般に知られているため，本稿では省略する．

3.3 整 合 性

ある性質 $p \in P$ が成立するかどうかは，状態に割り当てられている原始命題集合により決まる．よって，整合性を定義するためには，設計と実装の間の状態と原始命題集合の関係を決める必要がある．そこで，実装の状態が設計モデルの状態と同じであれば，実装においても対応する原始命題が成立するものと仮定する．設計と実装の双方に出現する変数に関する条件式などは，すべてこの仮定を満たすことになり，自然な仮定であると言える．

定義 3.4 状態の対応関係
状態の対応関係 ϕ を以下のような関数で表現する．

$$\phi : S_I \to S_D$$

定義 3.5 原始命題の同値性の仮定
設計モデルと実装において，以下の性質が成立するものと仮定する．

$$\forall s \in s_I . L_D(\phi(s)) = L_I(s_I)$$

定義 3.6 整合性 (性質の保存)
設計段階で検証した時相論理式の集合を P とすると，設計と実装の整合性は以下のように定義される．

$$\forall p \in P . (E||DM) \models p \Leftrightarrow (E||IM) \models p$$

$(E||DM) \models p$ と $(E||IM) \models p$ は，通常の線形時相論理と同様に定義されているものとする．
また，以下のような関係が成立する場合は，半整合性と呼ぶことにする．

$\forall p \in P.(E||DM) \models p \Rightarrow (E||IM) \models p$

3.4 対応関係と整合性

設計と実装が整合している，すなわち，実装において性質集合 P が成立するかどうかは，原始命題が成立するかどうかに依存しており，それは状態により決まる．また，原始命題の同値性の仮定を考慮すると，ϕ に相当するものを，実装に埋め込めば良いことがわかる．本研究では，実装の状態を直接観測するのではなく，ϕ を通して観測することにする．また，これらの定義からわかるように，設計モデルと実装が整合するかどうかは，ϕ と δ_I, λ_I に依存している．よって，ϕ 経由で観測できる状態に基づいて δ_I, λ_I が望ましくなっているかどうかテストすることが，本研究における設計と実装の整合テストである．以下では，モデル検査で保証する性質のみに注目する．すなわち，δ_I が望ましくなっているかどうかをテストする手法について考える．また，前述したように，整合テストでは，open system を対象としているが，本研究では，closed system を対象にする．よって，整合性は，closed system に対してモデル検査した結果と同じものになることである．

以下が形式的な定義と定理である．

定義 3.7 全準同型

$E||DM$ と $E||IM$ が準同型であることは，以下により定義される．

$$\forall a \in I. \forall s \in S_I. \quad \delta_{E||D}(\phi(s), a) = \phi(\delta_{E||I}(s, a)) \wedge \lambda_{E||D}(\phi(s), a) = \phi(\lambda_{E||I}(s, a)) \wedge \phi(s_{I0}) = s_{D0}$$

$E||DM$ と $E||IM$ が準同型である時，$HOM(DM, IM)$ と書くことにする．
また，ϕ が全射 (surjective function) である時，DM と IM は全準同型であると呼ぶ．
$E||DM$ と $E||IM$ は全準同型である時，$SHOM(DM, IM)$ と書くことにする．

定理 3.1 整合定理

原始命題の同値性の前提の下で，$E||DM$ と $E||IM$ が全準同型であるならば，それらの間に整合性が成立する．

$SHOM(DM, IM) \Rightarrow (\forall p \in P.(E||DM) \models p \Leftrightarrow (E||IM) \models p)$

証明

$E||DM$ における状態遷移列を $\sigma = \sigma_1\sigma_2 \cdots \sigma_i \cdots$ のように表現し，$\sigma_1 = s_{(E||D)0}$ から始まるすべての集合を Σ と書くことにする．同様に，$E||IM$ における状態遷移列を $\pi = \pi_1\pi_2 \cdots \pi_i \cdots$ のように表現し，$\pi_1 = s_{(E||I)0}$ から始まるすべての集合を Π と書くことにする．この時，(全) 準同型であることから，$\delta_{E||DM}(\phi(\pi_{i-1}), a) = \phi(\pi_i)$ が成立する．よって，Π 中の任意の状態遷移列 $\pi_1\pi_2 \cdots \pi_i \cdots$ に対応する，$\phi(\pi_1)\phi(\pi_2) \cdots \phi(\pi_i) \cdots$ は，$E||DM$ における状態遷移列でもある．すなわち，$\{\phi(\pi)|\pi \in \Pi\} \subseteq \Sigma$ である．ただし，$\phi(\pi) = \phi(\pi_1)\phi(\pi_2) \cdots \phi(\pi_i) \cdots$ である．
一方，DM における任意の状態遷移列 $\sigma_1\sigma_2 \cdots \sigma_i \cdots$ とすると，$E||DM$ と $E||IM$ が全準同型であるので，$E||IM$ において，$\sigma_i = \phi(\pi_i)$ となる状態遷移列 $\pi_1\pi_2 \cdots \pi_i \cdots$ が 1 つ以上存在する．すなわち，$\Sigma \subseteq \{\phi(\pi)|\pi \in \Pi\}$ である．これらのことから，$\{\phi(\pi)|\pi \in \Pi\} = \Sigma$ である．
ここで，原始命題の同値性の仮定より，$E||IM$ の任意の状態遷移列 π において，$\forall i. L_D(\phi_i(\pi_i)) = L_I(\pi_i)$ が成立する．よって，任意の $\pi \in \Pi$ に対して，$\pi \models p$ と $\phi(\pi) \models p$ は，同値である．そのため，$\{\phi(\pi)|\pi \in \Pi\} = \Sigma$ より，$(E||DM) \models p \Leftrightarrow (E||IM) \models p$ が成り立つ．

証明終

以上の関係は，抽象化[8] の枠組における完全な抽象化と同様のものである．また，実際には，設計モデルで成立する性質が実装でも保証されるだけで，その逆が保証されなくても良い場合がある．この場合，原始命題に関する仮定，および，$E||DM$ と $E||IM$ の関係を弱めることができる．これは，抽象化の枠組における，健全な抽象化と同様のものである．

定義 3.8 原始命題の健全性の仮定

以下が成立する時，設計モデルと実装の原始命題が健全であると呼ぶ．

$\forall s \in s_I. L_D(\phi(s)) \subseteq L_I(s_I)$

定理 3.2 半整合定理

原始命題の健全性の仮定の下で，$E||DM$ と $E||IM$ が準同型であるならば，半整合性が成立する．

$HOM(DM, IM) \Rightarrow (\forall p \in P.(E||DM) \models p \Rightarrow (E||IM) \models p)$

証明

整合定理の証明より，$E||DM$ と $E||IM$ 準同型であるなら，$\{\phi(\pi)|\pi \in \Pi\} \subseteq \Sigma$ が成り立つ．また，原始命題の健全性の仮定より，任意の $\pi \in \Pi$ に対して，$\phi(\pi) \models p \Rightarrow \pi \models p$ が成立する．これらから，$(E||DM) \models p \Rightarrow (E||IM) \models p$ である．

証明終

4. テストケースの自動生成

提案手法の概要を図 2 に示す．本研究では，設計モデルはモデル検査ツール Spin[3] の入

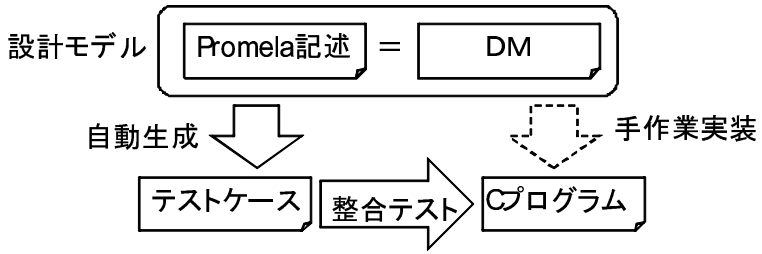

図 2 提案手法概要

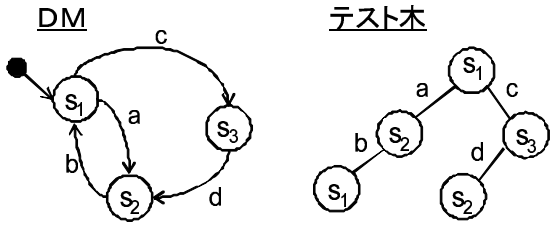

図 3 テスト木

力言語である Promela により記述されているものとする．この Promela 記述は，それ自身で Spin で検証することができる，すなわち，closed system であり，その正しさが検証済みである．また，DM と直接対応づけ可能なものしか記述しないものとする．テスト対象 (SUT, System Under Test) である実装は，任意の方法を取ることができる．ただし，内部状態を観測するための機構，すなわち，ϕ に相当するものは埋め込む必要がある．そして，設計モデルから整合性を保証するためのテストケースを自動生成する．ここで，生成されるテストケースは，設計モデルのモデル検査における環境に相当する部分である．そして，このテストケースを SUT に適用することにより，整合性を保証する．

4.1 設計モデルからテストケースの獲得

従来からの整合テストでは，SUT に入力を与え，期待する出力が得られるかどうかをテストする．本研究では，この従来からの整合テストとは異なり，実装の内部状態の一部を観測し，それが期待した状態になっているかどうかを調べる．期待した状態になっていれば，原始命題の同値性の仮定により，その状態で設計モデルと同じ原始命題が成立することになり，整合性が保証できるからである．よって，テストケースは与える入力と期待する出力，そして，期待する状態の 3 つ組の列により定義される．

定義 4.1 テストケース
テストケースの集合 TC は以下のような集合により表現される．

$$TC \subseteq seq\ (I \times O \times S_D)$$

ただし，$seq\ X$ は，集合 X の列の集合である．

整合テストでは，仕様のそれぞれの状態に到達可能な入力列に基づいてテストケースを作成する．このようなテストケースはテスト木 (testing tree) を用いて作成される．図 3 は，DM とそれから作成されるテスト木の簡単な例である．テスト木のノードは状態，枝は遷移に対応している．まず，DM の初期状態を木の根 (root) とする．そして，ノードが表現する状態から遷移可能な次状態がある時，その遷移を枝，次状態を 1 つ下のノードとする．もし，一番下のノードが表現する状態と，同じものを表現するノードが上にあれば，そのノードからは，それ以上，枝を張らない．このようにして作成されたテスト木の枝を走査することにより，それぞれの状態に到達可能でかつ，その状態に至るすべての入力列を求めることができる．

一方，Promela 記述において変数が使われていると，DM の状態は膨大になり，手作業でテスト木を作成するのは困難である．そこで，まず，Spin による状態探索アルゴリズムを応用してテスト木を作成する方法を提案した[5]．この方法では，Promela/Spin の埋め込み C 機能を用いて，状態探索の間，副作用的に探索情報を出力し，それに基づいてテスト木を作成する．埋め込み C 機能とは，Promela 記述中に，C コードの断片を記述することができ，それは，直接，状態探索のプログラム中に埋め込まれる．この C コードの断片において，探索に関する内部情報を参照して出力すれば，テスト木を作成するために十分な情報を獲得することができる．このテスト木に基づいて，入力と，期待する出力，状態の列を求めることができる．このようにして求められたテストケースの集合を TC_{pc} とする．

関連研究[4] においても，Spin を用いてテストケースを生成したと報告がある．しかしながら，Spin により生成される pan.c を変更して獲得したという説明があるのみで，詳細については触れられていない．提案手法は，系統的に Promela 記述の中に C コード断片を埋め込むだけで良いので，より容易であると考えられる．また，対象としたモデルも異なる．

4.2 実装の取り扱い

設計で検証した性質が実装においても成立するためには，原始命題が成立するかどうかに依存している．そして，その原始命題が成立するかどうかは，実装の状態がどのように変

化するかで決まる．そのため，実装の内部状態の変化を把握できなければ，設計と実装の整合性を保証することができない．そこで，本研究では，従来からの整合テストとは異なり，実装の内部状態を一部見ることができるものと仮定する．内部状態は，ϕ に相当するものを実装の内部に埋め込むことにより観測する．この埋め込みは原始命題の同値性の仮定が成立するように実現しなければならない．例えば，検証を行った Promela 記述において出現する変数は，そのまま実装でも用いて，その値を返せば，この制約を満たすことになる．これ以外にも，原始命題の同値性の仮定を満たす方法は考えられる．この埋め込みにより観測できる状態が，設計モデル DM と同じように遷移するのであれば，整合性を保証することができる．また，観測できる状態は安定していなければならない．すなわち，新たな入力記号を与えるまでは，状態は変化しないということである．もし，新たな入力記号を与えるまでに，異なる状態に変化するのであれば，網羅的にテストができない．

5. 実　　験

今年度は，以上のような手法で整合性が保証できるかどうか実験を行った．実験の対象は，日経エレクトロニクスにおいて紹介されていたキッチンタイマーの例である[6]．まず，紹介されていた状態遷移モデルを Promela により記述した．また，評価ボード用のソースコードもあったが，それでは実験を行うのに不便であったため，パソコン上で動作する，それと同様なプログラムを作成して実験を行った．

このキッチンタイマは調理時間などを計るためのタイマである．日経エレクトロニクスに付録として添付されていた 16bit マイコン搭載ボード「NE-R8C/25」上で動作する．このボードには，入力スイッチ 2 個，出力用の LED) が 2 個付いており，これらにより，キッチンタイマーの入出力を実現している．以下は，2 つのスイッチを，SW1 と SW2，2 つの LED を，LED1 と LED2 として参照する．キッチンタイマを起動すると，まず，SW1 により時間を設定する．計測時間は，SW1 を押すごとに更新され，LED1 と LED2 により 2 進数によりその時間が表現される．計測時間は 1 分〜3 分まで設定できる．そして，SW2 を押すことにより時間の計測が始まる．時間の計測中は，LED1 と LED2 は交互に点滅する．設定した計測時間が経過すると，LED1 と LED2 を同時に点滅させ，知らせる．その他にも，リセットや一時停止などの機能がある．キッチンタイマーは 3 つのタスク，キー入力監視タスク，計時タスク，表示タスクにより構成されている．計時タスクは，入力待ちや計測中などの状態を管理している．他の 2 つのタスクは，入力スイッチや LED を管理するためのものである．実際の時間の計測や LED 点滅間隔などは，1s，100ms，500ms の周期

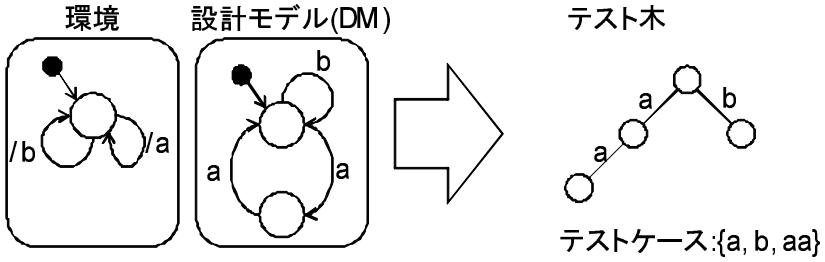

図 4　環境と設計モデル

ハンドラを用いて実現されている．

まずは，設計モデルとして状態遷移モデル DM を作成し，それに対応する Promela 記述を作成した．この Promela 記述の行数は 230 行である．パソコン上で動作する C のプログラムは 363 行であった．そして，提案したテスト木生成法を用いて，それぞれの状態に到達するテストケースの集合を作成したところ，1155 個のテストケースが生成された．

6. 考　　察

6.1 環境の取り扱い

すでに述べたが，テスト対象である DM はオープンシステム (open system)，すなわち，入力を与えると動作するというものに対してテストを行う．一方，モデル検査は，通常，環境と呼ばれる，対象への入出力の部分を作成して，閉じられたシステム (closed system) にして検査を行う．図 4 は，環境を追加して，設計モデルを closed system にしたものである．このように，環境では，DM への想定する入出力を記述する．Spin を用いたモデル検査では，このような closed system に対して，時相論理式が成立するかどうかを検査する．

この枠組では，3 つのテストケース生成のバリエーションが考えられる．

(1)　環境だけからテストケースを生成．
　　この場合，想定する入出力を網羅したテストケースが生成される．

(2)　設計モデルだけからテストケースを生成する．
　　従来からの整合テストと同じ方法である．この場合，可能な入出力をすべて考慮することができる．しかしながら，テストケースが膨大になる．また，一般には，入出力の間には依存関係がある場合が多く，それらを考慮していないテストケースには，意

味の無いもの，もしくは，想定していないものが含まれる可能性がある．

(3) 環境と設計モデルからテストケースを生成する．
本研究の場合である．この場合，環境として記述された想定する入出力に関して，設計モデルの内部状態を網羅したテストケースが生成される．環境の状態モデルに依存するが，通常は，2 の場合よりテストケースの数は少くなるはずである．すなわち，設計モデルの可能な入出力のうち，興味があるものに限定するということである．

それぞれの場合から生成されるテストケースの集合を TC_1, TC_2, TC_3 とすると，明らかに，$TC_1 \subseteq TC_3$ は成立する．設計モデルの内部状態を考慮するためである．実際，図 4 に関してテストケースを作成してみると，環境のみから生成されるテストケースの集合は，$\{a, b\}$ であるが，環境と設計モデルから生成した場合は，$\{a, b, aa\}$ となる．TC_3 と TC_2 は，一般には，比較不可能である．もし，環境の状態モデルが設計モデルより複雑である場合，TC_3 の方が数が多くなる可能性がある．例えば，環境の状態遷移モデルを図 5 とすると，TC_3 の数より，TC_2 の数の方が多くなる．しかしながら，TC_2，TC_3 により網羅する設計モデルの状態遷移は，明らかに，TC_2 の方が TC_3 より多い．つまり，TC_3 の数が，TC_2 より多くなる場合は，テスト対象の状態遷移の網羅の観点から，冗長なものを含んでいるのである．よって，環境は，興味があるものに限定するという目的で作成すべきである．

また，環境は検査したい性質により異なる．例えば，正常処理 (エラーを返さない動作) について検査したい場合と，異常対処処理 (エラーを返す動作) について検査したい場合は，入出力は異るはずである．よって，一般に，環境は複数作成され，それぞれの環境と設計モデルを組み合わせてテストケースの集合を求めることになる．現在の枠組では，環境の構成法については考慮していないが，本手法で生成するテストケースによりカバーできる設計モデル自体の整合性の範囲は，明らかに環境に依存している．現在は，$E||DM$ と $E||IM$ の整合性，すなわち，特定の環境 E の下での整合性を考えているが，理想的には，複数の環境を適切に構成することにより，DM と IM の整合性を保証できることが望ましい．

6.2 状態の対応関係と整合性

提案手法で生成したテストケースを用いれば，設計モデル $E||DM$ のそれぞれの遷移に対応する遷移が，実装 $E||IM$ に存在することは確認できる．また，ϕ の値域が設計モデルの状態に収まっていれば，全射であることも保証できる．しかしながら，実装 $E||IM$ のそれぞれの遷移に対応する遷移が，設計モデルにあるかどうかについては保証できない．これに関しては，もし，$\{\phi(s)|s \in S_{E||I}\} \subseteq S_{(E||D)}$ が成立するならば (テストがすべて問題なければ，全射であることがわかるので，$\{\phi(s)|s \in S_{E||IM}\} = S_{(E||D)}$ となる)，到達可能な状態において，任意の入力記号を 1 回入力してみて，その遷移先が正しければ良い．これは，従来の整合テストにおいて，遷移エラーを検出する場合と同様である．この前提条件の $\{\phi(s)|s \in S_{E||I}\} \subseteq S_{(E||D)}$ を成立させるための方法には，色々なものが考えられるが，基本的には，設計モデルに基づいて実装を行う際，制約を課すことにより実現できる．例えば，実装の状態遷移は，設計モデルに出現する変数にしか依存しないようにすることが考えられる．実装の状態に対応する設計の状態が存在しない場合，任意の入力記号を 1 回入力してみれば，設計の状態から逸脱しているかどうか判断できるため，遷移エラーを検出できると同時に，$\{\phi(s)|s \in S_{E||I}\} \subseteq S_{(E||D)}$ を保証できるのである．キッチンタイマーを用いた実験では，設計モデルと，ほぼ，1 対 1 に実装ができたため，この前提を満たすことができた．この例では，入力記号は 3 つあったので，整合性を保証するためには，3465 個のテストケースが必要であった．

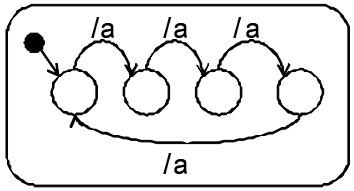

図 5 冗長な環境

しかしながら，より現実的には，$\{\phi(s)|s \in S_{E||I}\} \subseteq S_{(E||D)}$ のような前提を満たすのが困難な場合がある．一般に，実装の状態は設計モデルより多いことが予想され，その範囲を厳密に把握するのは困難だからである．図 6 は，設計モデルの 1 つの状態に対して，実装では，複数の状態が対応している場合である．この状況は，実装の際，新たな変数を導入するなどすると発生する．図では，設計モデルに出現する変数の値が v の時，新たに導入変数が $v'_1, v'_2, \cdots, v'_i, \cdots, v'_n$ のように変化している．ここで，遷移エラーを検出するためには，v'_i を変化させるトリガである $a_0, a_1, \cdots$ を認識する必要がある．そして，それぞれの状態に到達するテストケースを作成し，それらに任意の入力記号を 1 回付け加えなければならない．つまり，実装の際，新たに導入した変数の追跡が必要なのである．このような変数を直接的に取り扱うためには，実装の際，新たに導入した変数に関する情報を残しておき，それに基づいてテストケースを作成する方法が考えられる．しかしながら，その作業量は多いことが予想され，効率的なテストが行えない可能性がある．我々は，C プログラムの変

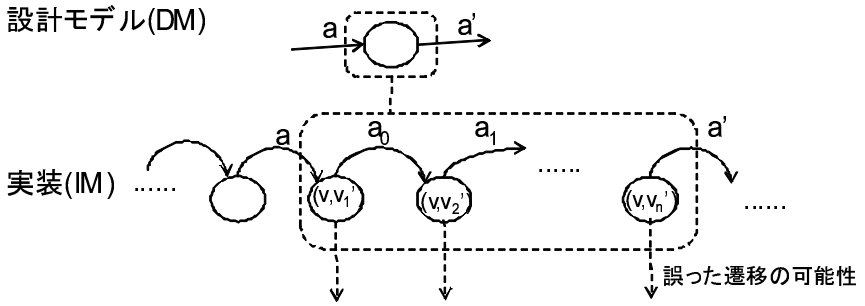

図 6 実装における状態の追加

数やメモリの一部を監視しながらモデル検査を行う実行時モデル検査手法について研究を行っている[7]．そこで，新たに追加した変数を監視して，実行時モデル検査により効率的に変数の追跡を行える方法について検討している．

6.3 抽象化と整合テスト

クリプケ構造を対象とした抽象解釈を用いた抽象化の枠組と，本研究における整合テストの枠組は，ほぼ同じである．一方，その使い方は異る．一般に，プログラムの抽象化では，与えられたプログラムに対して，健全もしくは完全な抽象化になるように，ϕ に相当する抽象化関数を決める．そして，抽象化関数を用いて抽象モデルを獲得，それに対してモデル検査を実行する．一方，整合テストでは，抽象モデルに相当する設計モデルとプログラムは与えられており，それらが抽象化の関係になっている，すなわち，抽象化関数 ϕ に関して健全，もしくは，完全になっていることをテストにより検査する．抽象化になっている ϕ が存在すれば，DM と IM は整合しているが，それを自動的に求めるのは困難である．

今回の実験では，DM に出現する状態と変数が実装にも出現するように実装したため，それらを ϕ により直接的に対応づけた．すなわち，$x_1, \cdots, x_n$ を双方に出現する変数，LB を双方に出現するラベルとすると，$S_D = S_I = V(x_1) \times \cdots \times V(x_n) \times LB$ であり，$\phi(x_1, \cdots, x_n, l) = (x_1, \cdots, x_n, l)$ である．ただし，$V(x_i)$ は変数 x_i の取り得る値の集合である．このように，設計の状態をどのように実装で考慮したかを考えると，ϕ が決まる場合が多い．しかしながら，一般には，複雑に関係している場合も考えられる．そこで，データマッピングや述語抽象など，抽象化手法で用いられている抽象化関数をヒントに ϕ を構成することも考えられる．

7. ま と め

本稿では，モデル検査により検証された設計モデルと実装が整合していることをテストする手法を提案した．従来の整合テストは，ブラックボックステストであり，テスト対象の入出力に関するテストにより，仕様と実装の状態が一致していることを保証しようとするものである．一方，本研究では，モデル検査による設計検証を前提としており，設計において保証された性質が，実装においても成立することが整合性である．この整合性を保証するためには，設計と実装の状態やそれに付随する原始命題に関して整合を取る必要があり，ブラックボックステストによる実現は困難である．そこで，実装の状態を確認できるという前提を置いて整合テストを行うための形式的な枠組，および，それに基づいた実験を行った．この枠組では，モデル検査により検証された設計を取り扱うため，従来の整合テストの枠組に，環境と原始命題の概念を導入し，整合性の定義，および，整合性を保証するための条件を示した．考察で示したように，この枠組では，環境の取り扱いと，状態の対応関係が重要である．これらについて，今後，研究を進める予定である．

参 考 文 献

1) D.Lee and M.Yannakakis: Principles and Methods of Testing Finite State Machines - a Survey, Proceedings of the IEEE, vol. 84, no. 8, pp. 1090–1123, 1996.
2) T.S.Chow:Testing software design modeled by finite-state machines, IEEE Transactions on Software Engineering, SE-4(3), pp.178–187, 1978.
3) G.J.Holzmann: The Spin Model Checker - Primer and Reference Manual, Addison-Wesley, 2004.
4) Rene G. de Vries and Jan Tretmans:On the fly Conformance testing using Spin., International Journal on Software Tools for Technology Transfer, 2(4), pp.382–393, 2000.
5) Nguyen Tam Thi Minh: モデル検査用記述に基づいたテストケースの生成法，修士論文，北陸先端科学技術大学院大学, 2009.
6) 穴田啓樹:組み込みアカデミー 2, 日経エレクトロニクス (連載記事), 2008.
7) 土肥雅俊, 青木利晃:C プログラムの実行に基づいたモデル検査実験, ソフトウェア工学の基礎ワークショップ, pp.87–92, 2008.
8) 田辺良則, 高井利憲, 高橋孝一:抽象化を用いた検証ツール. コンピュータソフトウェア, Vol.22, No.1, 2005, pp.2–44.