日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

情報制御システム記述モデルの検証用記述への変換と効率的検証

柳 翔太　小飼 敬　上田 賀一　大久保 訓　高橋 勇喜　中野 利彦

ソフトウェア品質を確保する手法に形式検証を上げることができる．しかし，実践レベルへの形式検証の適用は多くの問題を抱えている．本研究では，情報制御システムを対象とし，専用の記述言語で記述されたモデルに対する実用的な検証の実行方法について検討する．一般に，モデル検証はモデル検査ツールを使用することで実行されるため，以下の３点について検討する．まず，対象モデルを検証用記述に変換する必要があるが，変換には専門の知識が必要となり現場での運用が難しい．そのため，検証用記述への変換ルールを定義する．次に，検証の実行をサポートするために，設計した検証項目を LTL 式に変換する必要があり，その変換ルールを定義する．最後に，モデル検査では対象モデルの状態数が多いと実用的な範囲の時間で検証が実行できない場合がある．そのため，状態数を抑えるための工夫について検討する．モデルから検証用記述への変換方法と LTL への変換方法を定義し，実用化のための工夫を施した結果，効率的なモデルの検証を行うことができた．

1　はじめに

近年，システムの品質に対しての関心が高まっている．品質を高めるためには設計レビューやテスト・デバッグを行う必要がある．しかし，システムが大規模かつ複雑になるにつれて，従来の手法では十分に対応しきれなくなっている．設計レビューは人手による確認作業が基本なので，大規模システムにおいては人手によるレビューでは限界がある．また，テスト・デバッグについても，ツールによる自動化が進んでいるが，テストデータやテストケースの作成は基本的に人手で行っている [3].

このような問題を解決するための手段の一つに形式検証の一種であるモデル検査が挙げられる．モデル検査ではソフトウェアの仕様が意図したとおりであるか振舞いを網羅的に検証することができる．しかし，開発現場においてモデル検査を導入するためにはいくつか問題がある．

まず１つは，モデル検査を実行するためには対象のシステムを使用する検証ツールに対応するモデル記述言語で適切に記述する必要があることである．加えて検証したい性質を LTL(Linear Temporal Logic) 等の時相論理を用いて記述する必要もある．これらの記述には専門の知識を必要とするため導入へのハードルを上げる要因になっている．また，規模が小さいシステムであれば人手によって記述することも可能だが，規模が大きくなるにつれて人手による記述は困難になる．

現在，この問題に対して取り組んでいる研究は様々なものがある．例えば，高谷と新川 [7] は UML のシーケンス図からステートマシン図を経由し PROMELA への変換を行った．この研究では，確かに PROMELA への変換が行われたが，適用対象の規模が大きいとは言えず実用化に至るかどうかは不明である．また，藤原ら [1] は Struts アプリケーションの動作検証を目

Transformation of Information and Control System Description Model to Verification Model and its Efficient Execution.
Shota Yanagi, 茨城大学, Ibaraki University.
Kei Kogai, 茨城工業高等専門学校, Ibaraki National College of Technology.
Yoshikazu Ueda, 茨城大学, Ibaraki University.
Satoshi Okubo, 株式会社　日立製作所, Hitachi Ltd. .
Yuki Takahashi, 株式会社　日立製作所, Hitachi Ltd. .
Toshihiko Nakano, 株式会社　日立製作所, Hitachi Ltd. .

的として，PROMELA コードを自動生成するツールを作成した．実験ではツールを用いた検証を行うことができたが，検証対象が複雑になった際の検証時間に不安が残っている．金井ら [4] は UML 設計検証のための検証パターンの提案を行った．提案パターンに基づいて PROMELA と LTL を記述し検証を行うことができたため，提案パターンの有用性が証明された．しかし，対象の規模が小規模なため大規模なシステムに適用した場合にどのような結果になるかは不明である．

また，モデル検査では起こりえる状態を網羅的に検証するため，システムが大規模になると状態数が爆発的に増加して実用的な検証時間では検証を行うことができない問題がある [6]．検証の対象としたいシステムは大規模システムなので，この問題を解決しなければ現場へのモデル検査の導入は難しくなる．

そこで本研究では，情報制御システムを対象とし，専用の記述言語で記述されたモデルに対する検証を支援する方法を提案する．モデル検査の現場での実用化を進めるために，本研究では検証用記述への変換と状態数を抑えて検証時間を短くすることに焦点を当てることにする．対象モデルを検証用記述に変換する方法と LTL への変換方法については 3.1 で説明する．検証用記述への変換ルールを定義することで開発者はモデル記述言語等の専門知識を習得することなく検証用記述を得ることが可能になる．また，LTL への変換ルールを定義することで LTL の知識を必要とせずに検証を実行できるようになる．

今回，モデルから検証用記述への変換方法と LTL への変換方法を定義した結果，対象としたモデルの検証を行うことができた．

2 情報制御システム記述言語について

本研究で対象としている情報制御システム記述言語について説明する．

2.1 記述言語の概要

今回対象とする情報制御システム記述言語は，情報制御システムを対象として開発されているモデル記述言語である．システムの振舞いや実行条件はそれぞれ専用の記法によってモデル化され，複数のモデルを組み合せることでシステム全体を表現している．

2.2 モデルの構成要素

モデルは以下のような要素から構成されている．

- オブジェクト
  システムを構成する設備を表す．設備の状態等の属性とその属性値を持つ．
- アクタ
  オブジェクトで定義した設備が持つ属性値の参照や更新などの操作を表す．アクタはシステムの振舞いや制約条件をルールベースのモデルとして表現している．
- アクタシナリオ
  アクタの呼び出し順序を表す．

2.3 モデルの記法

情報制御システム記述言語では，オブジェクト，アクタ，アクタシナリオが以下の 5 種類の記法を利用してモデル化される．

- 制御判断
  制御条件と，その条件が成立した際に実行する操作の組の集合を表す．制御条件は条件定義で定義されたものを使用し，実行する操作は操作定義で定義される操作になる．
- 条件定義
  制御条件の定義を表す．制御条件は他の制御条件との依存関係も表現する．制御条件はオブジェクトの属性値を組み合せて表現される．
- 操作定義
  制御判断において制御条件が成立時に実行する操作を定義する．操作は，操作の条件と条件成立時に実行する操作内容の組で表す．条件と操作内容の表記については，制御判断と同様である．
- オブジェクトスキーマ
  オブジェクトが持つ属性一覧と他のオブジェクトとの依存関係を表す．
- 検証項目
  制御判断において，検証したいシステムの状態を表す．

制御判断	
条件	実行項目
設備A	設備B
属性A	−
属性値A	実行項目A
属性値B	−

図 1　制御判断

3　PROMELA 記述への変換

本研究では検証ツールとして，モデル検証ツール SPIN を使用する [2]．SPIN は PROMELA と呼ばれるモデル記述言語を用いて記述されたモデルを入力として検証を行う．PROMELA 記述への変換ルールを 2.3 で説明した記法を対象に定義する．各記法ごとに定義した変換ルールを 3.1 で説明し，実用化のために施した工夫について 3.2 で説明する．

3.1　変換ルールの定義

3.1.1　制御判断の変換

制御判断の内容は Control プロセスと命名したプロセス内に記述される．図 1 に変換元の制御判断を示し，図 2 に変換後の PROMELA 記述を示す．制御判断の制御条件と実行項目の組を

属性名 == 属性値 − > 実行フラグ = true

というように変換する．記号「==」は等号,「->」は左辺が実行されたなら右辺を実行することを意味している．属性の属性値が条件と一致すると実行項目の操作を実行可能にするフラグを True にする処理を行う記述になっている．制御条件と実行項目の組を順番に do 文（ループ文）中に配置し，上から順番に制御条件の成立/不成立をチェックする．制御条件が成立した場合は先頭に戻り，不成立の場合は次の制御条件をチェックする．一度でも制御条件が成立するか，最後まで制御条件が成立しなかったときに次のプロセスへプロセスの実行権限を渡す．

3.1.2　条件定義の変換

条件定義の内容は Update プロセスと命名したプロセス内に記述される．図 3 に変換元の条件定義を示し，図 4 に変換後の PROMELA 記述を示す．条件定義の属性定義と条件の組を

```
active proctype Control(){
do
::Ln == 1 && ControlTurn == true;
  if
  ::d_step{Attribute_A == AttributeValue_A
           -> Execute_A = true ->
           ControlTurn = false ->
           ExecuteTurn = true -> Ln = 1}
  ::else -> Ln = 2
  fi
::Ln == 2 && ControlTurn == true;
  if
  ::d_step{Attribute_A == AttributeValue_B
           -> skip ->
           ControlTurn = false ->
           ExecuteTurn = true -> Ln = 1}
  ::else -> d_step{Ln = 1 ->
            ControlTurn = false;
            UpdateTurn = true}
  fi
od
}
```

図 2　PROMELA による制御判断の記述

更新条件 − > 属性 = 属性値

というように変換する．更新条件は属性更新フラグと条件定義で定義されている条件で構成されている．属性更新フラグは属性が更新されていることをチェックするためのフラグである．属性定義と条件の組を条件定義モデルごとに if 文内に記述する．モデルごとの if 文をすべて Update プロセス内に配置する．Update プロセスは，更新条件が成立すると属性値の更新を行い，属性更新フラグを True にし，更新条件内の属性更新フラグを False にする．更新可能な属性値が存在しなくなったとき，次のプロセスに実行権限を渡す．

3.1.3　操作定義の変換

操作定義の内容は Execute プロセスと命名したプロセス内に記述される．図 5 に変換元の操作定義を示し，図 6 に変換後の PROMELA 記述を示す．操作定義の条件と実行項目の組を

実行条件 − > 属性 = 属性値

というように変換する．実行条件は実行フラグと操作定義で定義されている条件で構成されている．条件と

条件定義		
属性定義		条件
属性名称	属性値	設備A
		属性B
属性A	属性値A	属性値C
		属性値D
	属性値B	属性値E

図 3　条件定義

```
active proctype Update(){
do
::UpdateTurn == true;
  if
  ::d_step{Attribute_B_UpFlag == true &&
    (Attribute_B == AttributeValue_C ||
     Attribute_B == AttributeValue_D) ->
     Attribute_A = AttributeValue_A ->
     Attribute_A_UpFlag = true ->
     Attribute_B_UpFlag = false}
  ::d_step{Attribute_B_UpFlag == true &&
     Attribute_B == AttributeValue_E ->
     Attribute_A = AttributeValue_B ->
     Attribute_A_UpFlag = true ->
     Attribute_B_UpFlag = false}
  ::else -> skip
  fi
od
}
```

図 4　PROMELA による条件定義の記述

実行項目の組を条件定義モデルごとに if 文内に記述する．モデルごとの if 文をすべて Execute プロセス内に配置する．Execute プロセスは，実行条件が成立すると属性値の更新を行い，属性更新フラグを True にし，実行フラグを False にする．実行可能な属性値が存在しなくなったとき，次のプロセスに実行権限を渡す．

3.1.4　オブジェクトスキーマの変換

オブジェクトスキーマに記述されている属性と属性値は，mtype と呼ばれるデータ型を用いてグローバルに宣言する列挙型の変数と定数に変換する．

変換例

操作定義		
条件		実行項目
設備A	設備B	設備B
属性A	属性B	属性B
属性値A	属性値C	属性値E
属性値B	属性値D	属性値F

図 5　操作定義

```
active proctype Execute(){
do
::ExecuteTurn == true;
  if
  ::d_step{Execute_A == true &&
    Attribute_A == AttributeValue_A &&
    Attribute_B == AttributeValue_C ->
    Attribute_B = AttributeValue_E ->
    Execute_A = false ->
    Attribute_B_UpFlag = true}
  ::d_step{Execute_A == true &&
    Attribute_A == AttributeValue_B &&
    Attribute_B == AttributeValue_D ->
    Attribute_B = AttributeValue_F ->
    Execute_A = false ->
    Attribute_B_UpFlag = true}
  ::else -> skip
  fi;
  if
  ::d_step{Execute_A == false ->
           ExecuteTurn = false;
           UpdateTurn = true}
  ::else -> skip
  fi
od
  }
```

図 6　PROMELA による操作定義の記述

```
mtype = 属性値
mtype 属性名 = 属性値
```

属性はグローバル変数として定義されているので，どのプロセスからも同じ属性に対して参照や属性値の更新を行うことができる．

3.1.5　LTL 式への変換ルール

検証項目を LTL 式へ変換するためのルールを説明する．現在定義している変換ルールは図 7 に示す検証項目に対応する 2 つのパターンがある．なお，図 7 は説明しやすいように簡略化した検証項目となって

検証項目		
条件		結果
条件A	条件B	−

検証項目		
条件		結果
条件A	条件B	結果A

図 7　検証項目

いる．

- パターン 1
  条件 A と条件 B が同時に成立してしまう場合（図 7 上の検証項目）
- パターン B
  条件 A と条件 B が成立するならば結果 A も成立してしまう場合（図 7 下の検証項目）

パターン 1 では各条件を「&&」で結び「()」で囲むことにする．そして,「()」の左側に”いつかは”を意味する時相演算子「<>」を付加する．これにより,「いつかは各条件が同時に成立する」場合に真となる LTL 式に変換することができる．変換例は次のようになる．

<>(条件 A && 条件 B)

パターン 2 の場合は，検証項目中の条件を「->」の左辺に記述し，結果を右辺に記述するようにする．結果には時相演算子「<>」を付加する．これにより,「条件が成立するといつかは結果が成立する」場合に真となる LTL 式に変換することができる．変換例は次のようになる．

条件 A && 条件 B − ><> 結果 A

両方のパターンにおいて，条件は 1 つ以上であればいくつでも良い．パターン 1 において条件を増やす場合は「&&」を用いて条件を増やす．パターン 2 において条件を増やす場合は，「->」の左辺に追加する．

3.2　実用化のための工夫

モデル検査を行う際に問題となるのが状態爆発の問題である．モデル検査では取りうる状態の組み合わせを網羅的に探査するため，状態数が増加するにつれて状態の組み合わせが爆発的に増加し，実時間内での検証が不可能になってしまう．本研究の目的は実際の現場でのモデル検査の普及であるためこの問題は解決しなくてはならない．そこで，PROMELA 記述へ変換する際に状態爆発を抑えるいくつかの工夫を施している．

3.2.1　非決定的選択の排除

PROMELA では非決定的選択が許されている．非決定的選択とは，複数の実行可能な処理が存在するときその中からランダムに 1 つの処理を選択して実行することである．非決定的選択があると状態の遷移がランダムになるため状態の組み合わせが多くなる．そのため，検証時間の増加や状態爆発を起こす要因になる．

そこで，PROMELA 記述から可能な限り非決定的選択を排除するために，条件判断から処理の実行までの一連のステップを不可分操作として記述する．不可分操作にすることで一連の処理を想定した順番通りに実行することができる．また，一連の処理の間に他のプロセスの命令が割り込むことを防ぐことができる．これにより，割り込みを考慮しなくてよくなるため状態の組み合わせを減少できる．

3.2.2　PROMELA における外部環境の表現

適切な検証を行うには外部からの入力等の外部環境を PROMELA で表現する必要がある．対象となる情報制御システム記述言語では外部からの干渉として属性値の変更が考えられるので外部入力を表現する．現実的には外部入力の値は決まった特定の組み合わせが与えられるわけではない．そのため，外部入力を忠実に表現すると常に入力値がランダムになる．しかし，入力値がランダムに与えられると検証の際に探査する状態の組み合わせが増加してしまう．これは，検証時間の増加を引き起こすため検証の実用化を考慮したときに問題になる．

そこで，入力値の組み合わせとして考えられる全ての組を用意し，一定間隔ごとに一組ずつ入力として与えることにする．外部入力は Input プロセスと命名したプロセス内に記述することにする．これにより，PROMELA 記述から非決定的な部分がなくなり状態の組み合わせの増加を防ぐことができる．

3.2.3 プロセスの実行順序について

PROMELA 記述に変換する際に 3.1 の定義では，すべてのプロセスは常にアクティブな状態であるように宣言しているため，通常ではどの順番にプロセスが実行されるか分からない．そのため，意図しない動作が起きる可能性を排除するためにプロセスの実行順序をフラグによって管理する．また，実行順序を管理することでプロセスの非決定的な実行をなくすことができるため，状態遷移数の増加を防ぐことができる．

自身に実行権限があるときにプロセスは実行され，実行後に自身の実行権限を放棄し次のプロセスへ実行権限を譲渡するようにする．これにより，プロセスは常にアクティブな状態だが実際に実行されているプロセスは常に一つだけになる．プロセスの実行順序は，Control プロセス，Execute プロセス，Update プロセス，Input プロセスの順にする．

3.2.4 Update プロセスにおける属性値更新方法

Update プロセスにおいて，更新されている属性値が関係する箇所のみ処理を行うようにする．これにより，属性値が更新されていないのに処理を行う無駄を排除することができる．

4 適用実験

定義した変換ルールを用いて検証を行うことが可能であることを証明するために実験を行った．実験内容は，定義した変換ルールをモデルに適用し手動で PROMELA 記述へ変換を行い，検証ツールを使用して検証を行うものとする．

検証する PROMELA 記述は，3 章で定義したルールを適用したものの他に，研究の初期段階で定義したルールを適用した場合の記述も比較対象として使用する．記述パターンは以下の通りになっている．

- 方式 A
  各条件定義モデルと各操作定義モデルをそれぞれ 1 つのプロセスにまとめずに，モデルごとにプロセスを作成する方式
- 方式 B
  各条件定義モデルと各操作定義モデルをそれぞれ 1 つのプロセスにまとめて，各プロセスの実行順序を制御しない方式
- 方式 C
  各条件定義モデルと各操作定義モデルをそれぞれ 1 つのプロセスにまとめて，各プロセスの実行順序を制御する方式

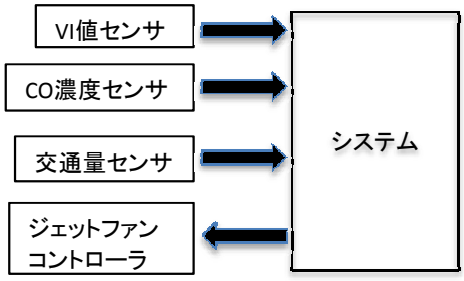

図 8 トンネル換気制御システム構成図

4.1 対象モデル

対象となるモデルは，トンネル内の換気状態の調整を行う「トンネル換気制御システム」を情報制御システム記述言語でモデル化したものである．

システムの構成を図 8 に示す．システムはトンネル内に設置された 3 つのセンサからデータを受信する．その後，受信したデータを基に換気ファンの出力値を決定しジェットファンコントローラに送信する．ジェットファンコントローラは換気ファン出力値に応じてファンの制御を行う．これらのシステムの振舞いをモデル化したものの一部を図 9 に示す．

4.2 検証内容

「トンネル換気制御システム」に対する検証項目を 3.1.5 で定義したルールを用いて LTL 式に変換し，検証ツール SPIN を用いて検証を行った．設計した検証項目は 2 つありそれぞれ検証項目 1 と検証項目 2 とする．

検証項目 1 は「要求ありと現状値が同時に成り立ってしまう」ということを検証するものである．検証項目 2 は「要求なしが成立したならば換気ファン出力更新も成立してしまう」ということを検証する．LTL 式はそれぞれ次のようになる．

検証項目 1

制御判断

ルール	
条件	実行項目
トンネル	ジェットファン コントローラ
換気ファン出力更新要求	–
要求あり	換気ファン出力更新
要求なし	–

条件定義

ルール		
属性定義		条件
属性名称	属性値	トンネル
		換気ファン出力制御
換気ファン出力更新要求	要求あり	パワーダウン
		パワーアップ
	要求なし	現状値

操作定義

ルール		
条件		実行項目
トンネル	ジェットファン コントローラ	ジェットファン コントローラ
換気ファン出力制御	出力値	出力値
パワーダウン	最大	レベル4
パワーダウン	レベル4	レベル3
パワーダウン	レベル3	レベル2
パワーダウン	レベル2	レベル1
パワーダウン	レベル1	停止
パワーアップ	停止	レベル1
パワーアップ	レベル1	レベル2
パワーアップ	レベル2	レベル3
パワーアップ	レベル3	レベル4
パワーアップ	レベル4	最大

図 9　トンネル換気制御システムモデル

<>(p && q)

PROMELA 記述中では「p を要求あり」と定義し，「q を現状値」と定義する．

検証項目 2

p − ><>q

PROMELA 記述中では「p を要求なし」と定義し，「q を換気ファン出力更新」と定義する．

今回対象としたモデルは，検証項目 1 を検証するとモデルの欠陥が発見されるように設計している．また，検証項目 2 では欠陥が発見されないように設計した．

表 1　検証項目 1 の検証結果

	作成状態数	探査遷移数	反例の有無	検証時間
方式A	検証不可	検証不可	検証不可	検証不可
方式B	1262	2016	あり	0.11秒
方式C	844	844	あり	0.07秒

表 2　検証項目 2 の検証結果

	作成状態数	探査遷移数	反例の有無	検証時間
方式A	検証不可	検証不可	検証不可	検証不可
方式B	3341	5535	なし	0.62秒
方式C	1272	1274	なし	0.12秒

4.3　検証結果

SPIN による探査の深さを 100 万に設定し検証を行った．検証項目 1 と検証項目 2 の検証結果をそれぞれ表 1 と表 2 に示す．

検証項目 1 に対する検証では方式 A は状態数が膨大になってしまい検証を行うことができなかった．方式 B と方式 C は検証を行うことができ，両方とも反例がありモデルの欠陥を発見することができた．作成した状態数と探査遷移数は方式 C の方が両方とも少ない結果になった．検証項目 2 に対する検証では方式 A は検証項目 1 に続いて状態数が膨大になってしまい検証を行うことができなかった．方式 B と方式 C は検証を行うことができ，両方とも反例がなかったため検証項目が成立しないことを証明した．．検証項目 2 の場合も検証項目 1 と同様に方式 C の方が作成した状態数と探査遷移数が少なくなっている．また，方式 B と方式 C の検証時間は方式 C の方が早く，いずれも 1 秒以下であった．

これらのデータについて 4.4 で考察する．

4.4　考察

変換ルールをモデルに適用して検証を行った結果について考察する．

4.4.1　状態数について

まず，それぞれの方式で作成された状態数について考えてみる．方式 A では複数のプロセスを乱立させ

た上に各プロセスに対して何も制御をしなかったため，状態の組み合せが膨大なものになってしまった．一方で方式 B と方式 C ではプロセスをモデル種別ごとに統一したため，状態の組み合わせは少なくなり検証を行うことができた．さらに，方式 B と方式 C を比較すると方式 C の方が作成状態数が少なくなっている．これは，方式 C においてプロセスの実行順序を制御したためである．プロセスの実行順序を制御したために非決定的選択が排除され，状態の組み合わせが少なくなったと思われる．

次に，探査遷移数について方式 B と方式 C を比較してみる．探査遷移数は方式 B より方式 C の方が少なくなっている．これには，作成した状態数の差が影響しており，状態数が少なければ遷移する数も自ずと少なくなる．また，方式 C の探査遷移数は作成された状態数とほぼ同じになっていることが分かる．これは，方式 C の場合は可能な限り非決定的選択を排除しているため，遷移の道筋がほぼ定まっているためである．

4.4.2 実用性について

モデル検査を現場で使用する際の問題の一つである検証時間について考える．検証時間は，実験結果を見ると今回対象にしたシステムの規模でなら問題なく実時間内に収めることができる．これは，状態数を可能な限り減少させる工夫を施したためである．

次に，モデル検査を現場に導入する際の手間について考える．今回検証用記述への変換ルールを定義したことで，モデルから検証用記述への変換が機械的に行えるようになった．また，検証項目から LTL 式への変換ルールの定義も行った．これらを定義したことにより，将来的にはモデルと検証項目を作成するだけで自動的に検証を行うことが可能になる．

検証自動化の例としては小池 [5] の研究が挙げられる．小池は，状態遷移表と付加的な表から制約条件まで含む検査プログラムを自動生成するモデル検査支援環境をした．この研究では，実際にソフトウェア開発プロジェクトに適用し評価した結果，効果が出ているので検証を自動化することができれば現場での導入も進むと思われる．

4.4.3 反例の分析方法

SPIN を用いて検証を行い欠陥を発見すると，反例として欠陥に至る経路をファイルに出力する．このファイルを基に欠陥の原因を分析する．しかし，反例ファイルの記述内容は専門知識がないと理解することが難しい．また，対象のモデルに関しての知識を持っていないと欠陥の原因を特定することは困難になる．SPIN の知識と対象システムの知識を持っている者が反例の分析を行えば．欠陥の原因を特定することができると思うが，確実性やコストの面から望ましいとはいえない．そのため，誰でも容易に反例の分析を行える方法を考える必要がある．現状で考えられる方法としては欠陥原因のパターン化である．過去の反例ファイルの解析結果を基に欠陥原因をパターン化し，反例ファイルに適用することで欠陥原因を特定する支援ができるのではないかと考える．

反例の分析方法を確立することはモデル検査の実用化を考える上で外せない問題である．今後はこの問題に力を入れていく必要がある．

5 おわりに

本研究では現場でのモデル検査の普及を支援するために，モデルから検証用記述への変換ルールの定義と，検証項目から LTL 式への変換ルールの定義を行った．その結果，モデルに変換ルールを適用し，検証を行うことができた．また，検証時間を短くするために検証時に作成される状態数を減少させる工夫をし，状態数の減少と検証時間を短くすることに成功した．

SPIN を用いた研究は，本研究で用いた表ベースのモデルを PROMELA に変換する研究の他にも様々なものがある．例えば，山口ら [8] はワークフローネットと呼ばれるペトリネットを PROMELA に変換し SPIN を用いて検証する研究を行った．検証の結果，ペトリネットのまま検証を行うよりも SPIN を用いて検証した方が検証時間が短いという結果がでた．この研究では SPIN を用いた方が良い結果になったが，逆に SPIN で検証した方が時間がかかったり場合によっては SPIN では検証できないということもあるだろう．SPIN に適している部分は SPIN で行い，適

していない部分は別の手段を用いることもモデル検査の実用化には必要なことだと考える.

今後は, 反例の分析方法の検討やモデルからの変換を自動的に行う検証ツールの作成を行うことにする.

参 考 文 献

[1] 藤原貴之, 岡野浩三, 楠本真二：SPIN による Struts アプリケーションの動作検証を目的としたモデル生成手法の提案, 電子情報通信学会技術研究報告, SS, ソフトウェアサイエンス Vol. 105, No. 491, 2005, pp. 73–78.

[2] Holzmann, G. J.：The Spin Model Checker: Primer and Reference Manual, Addison-Wesley, (2004).

[3] 荻谷昌己 監修：SPIN による設計モデル検証, 近代科学社 (2008).

[4] 金井勇人, 岸知二：UML 設計モデル検査技術のための検証パターンの提案, 情報処理学会研究報告, ソフトウェア工学研究会報告 Vol. 2006, No. 48, 2006, pp. 17–24.

[5] 小池隆：状態遷移表に基づくモデル検査の支援環境, 情報処理学会研究報告, EMB, 組込みシステム Vol. 2008, No. 116, 2008, pp. 91–96.

[6] Mordechai B. A.：Principles of the Spin Model Checker, Springer(2008).

[7] 高谷彰俊, 新川芳行：UNL シーケンス図におけるモデル検証方法, 電子情報通信学会技術研究報告, SWIM, ソフトウェアインタプライズモデリング Vol. 108, No. 56, 2008, pp. 13–18.

[8] 山口宗師, 山口真悟, 田中稔：モデル検査ツール SPIN によるワークフローネットの健全性の判定について, 電子情報通信学会技術研究報告, CST, コンカレント工学 Vol. 107, No. 203, 2007, pp. 7–12.