Vol.2016-SE-191 No.20
2016/3/15

モデル変換によるドメイン固有性質の形式検証に関する研究

須藤 一輝[2]　小野 康一[1]　深澤 良彰[2]

概要：UML による Web アプリケーションモデルをモデル検査系で形式検証する一手法を提案する．このような目的の既存手法では，モデル変換により時間オートマトンなどの形式表現を得てそれを検証するものがあるが, 実際のソフトウェアモデルは規模と複雑さが大きいため，単純な変換では状態爆発により検証が現実的には不可能である．そこで，アプリケーションドメイン固有の機能や役割の情報をモデルに記述しておき，それを手がかりとして簡略化された時間オートマトンに変換することで状態数を削減する方法を提案する．その代償としてドメイン固有の性質のみが検証対象となるが, そのような性質はドメイン内のアプリケーション全般において共通に存在するので有用と考える．本稿では手法と適用事例について述べる．

Sudo Kazuki[2]　Ono Kouichi[1]　Fukazawa Yoshiaki[2]

1. はじめに

近年, ソフトウェアの開発は大規模化・複雑化の一途を辿っている．一方, ソフトウェアの開発期間は短縮されている．よってソフトウェアを適切に開発することは困難である．そのような要求に対し, 一解決手法としてモデル駆動開発による抽象化が挙げられる [11].

モデル駆動開発においては, ソフトウェアを抽象的なモデルで表し, コード生成を行う．これによりソースコード中心な設計から離れることが出来る．その場合, モデルとして統一モデリング言語による UML モデル [1] や SysML モデル [2] が主として用いられる．しかし，抽象的なモデル表現を用いてコード中心開発から脱却できるとしても，モデル自身は人間が記述するため誤りが入るおそれは依然として残る．モデルの誤りをその記述工程で除去せずに下流工程で用いれば誤りの発見が遅れて手戻りによる開発コスト増大の原因となる．このため，モデルの記述時点における検証の必要がある．モデルの検証には一手段として形式検証技法が用いられる．形式検証技法は数理論理学を基に, 厳密な検証を行うことが可能である．特にその中でもモデル検査技法は産業界で注目を集めている [12][21][22][24]. モデル検査技法ではシステムの状態空間を網羅的に探索する．また, 開発の上流工程から検証可能であり, 早期に欠陥を発見し円滑な開発およびコスト削減が見込める.

このようなモデル検査技法をモデル駆動開発において利用する研究が提案されている [13][25]. しかし, 現実の開発において記述されるモデルは検証したい性質との関連を持たない情報を含んで複雑になっている．また, 状態が詳細化されて規模が巨大になっていることが多く, 形式検証技術を直接適用すると探索空間が巨大になってしまうことが多い. ゆえに, 現実的なシステム開発において, モデル検査技法を直接に適用することには技術的課題が存在している. よって, 形式検証技術に適用するためにはモデルから検証したい性質と関連を持たない情報を省いたり, 状態を簡略化する等の抽象化を行う必要がある．しかし, 検証したい性質に関わるアプリケーションドメイン固有の知識が無ければ, 適切な抽象化が行えない. また, 知識を有した人間であれば適切な抽象化が見込めるが, 機械的では無く, 形式検証技術の利点が損なわれてしまう.

このような技術的課題に対し, 本研究ではアプリケーションドメイン固有の性質を用いて, モデルを変換し検証を行う手法を提案する．具体的にはアプリケーションドメインにおける役割などを表現する UML プロファイルをあらかじめ定義し, それを適用して記述したモデルから形式検証技術が適用可能な形式へ変換し, 検証を実現する．また, 本研究ではアプリケーションドメインとして在庫管理系のアプリケーションを選択した．在庫管理系のアプリケーショ

[1] 日本アイ・ビーエム株式会社 東京基礎研究所
IBM Research - Tokyo
[2] 早稲田大学
Waseda Uniersity

ンはオンラインショッピングシステム, ホテル予約システム, 航空便予約システム等の幅広いシステムに関わりがあり, 検証を行うことに意義がある. なお, 本研究で提案する手法は在庫管理系以外のアプリケーションドメインに対してもその UML プロファイルに対応する変換規則を定義することで適用可能である.

本論文の構成は以下のとおりである. 第 2 章では研究の背景に触れ, その問題点ついて述べる. 第 3 章では本研究の提案手法について概要を述べる. 第 4 章では本手法の詳細と手順について述べる. 第 5 章では提案手法を実際に適用した結果について述べる. 第 6 章では関連している研究について述べ, 第 7 章で本論文の総括とする.

2. 研究の背景

この章では, 形式手法の概要を述べた上で, その一領域であるモデル検査技法について説明する. さらに, ソフトウェアモデルやシステムモデルを元にモデル検査技法を応用して, 形式的に検証する既存技術の問題点について述べる.

2.1 形式手法概要

形式手法は集合論や論理学等の数学体系の知識に基づいて, 厳密な検証や証明を行う技術である.

形式手法は主に形式仕様記述, 定理証明系, モデル検査系等の技術によって構成されている. 本稿では形式検証技術の一つであるモデル検査系を用いた手法を提案する.

2.2 モデル検査技法概要

モデル検査技法はシステムの取りうる動作を状態モデルとして表現し, その状態空間を網羅的に探索することによって, 与えられた性質が満たされるかを判定する検証技術である. オートマトン, 時相論理, グラフアルゴリズム等の数学的な概念で構成された技術である [20]. しかし, 他の検証技術に比べ, 必要とする数学的知識が少なくて良いこともあり, 近年産業界で注目を集めている [12].

モデル検証系に与える入力として, 検証したい性質とシステムの動作を表すオートマトンがある. 検証する性質として表 1 に示す様な一般的な性質を選ぶことが多い. このような一般性的な質は基本的に要求仕様を基にして, モデル検査系に適用するために時相論理式の形式で表現する. 検証系にこれらの入力を与えると自動的に網羅的検索を行うことで検証する. 検証系の出力として検査した性質を満たしているのか否かを表す検証結果や満たされない場合の実行系列を示した反例がある. 得られた反例を解析することで, 検証したモデルの初期状態からどのような経緯で検証したい性質を満たさなくなったのか確認することができる.

モデル検査技術の主な問題点として状態爆発や時相論理式の記述がある. 現実的な開発において用いられる様な規模が大きいモデルをモデル検査技術に適用した場合は, モデル検査系の入力として与えた状態が多くなってしまう可能性が高い. よって, このモデルに対して網羅的に検索しようと試みると, 実行経路数が膨大になってしまい検証が出来ない. このような問題を状態爆発問題と呼ぶ. 一方, 時相論理式に関してはオートマトン記述に比べ, 記述の敷居が高いという問題がある. オートマトンはステートマシーン図に似ている部分もあり, ステートマシーン図を描く技術者であれば記述は難しくない. しかし, 時相論理式は自然言語によって記述された要求仕様から, あらゆるシナリオに対して成り立つような性質を, 静的かつ宣言的に記述する. よって通常の開発と比較するとギャップが生じてしまう.

性質	説明
到達可能性	初期状態からある特定の状態へ到達する可能性があること
安全性	ある正当でない状況に陥ることが決して怒らないこと
活性	将来においていつか必ず, ある特定の状況がおこること
公平性	ある特定の状態が起きる状態があればいつか起こること

表 1 一般的な性質

代表的なモデル検査ツールとしては SPIN[3], SMV[4], UPPAAL[5] 等がある。SPIN は米国のベル研究所で開発されたツールである. SPIN では Promela と呼ばれる言語によって, 検査対象のモデルを記述する. Promela の文法は C 言語に類似している部分が多く, C コードの埋め込みも行える. SMV は米国の CMU で開発されたモデル検査ツールである. BDD 法をベースとした最初のモデル検査ツールである. SMV では CTL で記述した仕様を検証することが出来る. UPPAAL はスウェーデンの Uppsala University とデンマークの Aalborg University によって開発されたツールである. UPPAAL ではリアルタイムシステムのモデリング及びシミュレーションを行う. 他のツールと違いグラフィカルなエディタで時間オートマトンを記述する.

2.3 既存研究の問題点

一般にモデルを用いてソフトウェアを開発する際, UML あるいは SysML 等の統一モデリング言語によって開発対象を記述することが多い. 特に, UML モデリングツールでは下流工程のモデルからソースコードを生成する機能が備えられ, 開発の効率化に寄与している. しかし, UML 等のモデルに対して直接的に形式検証技術を適用することは非常に困難である. 形式検証技術を適用するためには対象言語の意味が厳密に定義されている必要がある. しかし, UML や SysML は言語仕様が非常に巨大であるにも拘らず, 意味の厳密性を欠く部分を多く有している. 特に, UML 1.x はスケッチとしてのモデリングが主眼であったため, 言語全般にわたって意味が曖昧だった. その後, UML 2.0 が定義される際に Executable UML の概念が持ち込まれたことにより, 実行に関係する部分の意味が明確化された. それによって, UML 2.x の言語仕様の一部は自然言語に依る

定義ながら意味の厳密性が向上している．また, SysML 1.x は UML 2.x をベースとした派生言語なので同様である．そのような状況に基いて, 既存研究では UML 2.x や SysML 1.x のうち, 意味が厳密になっている部分のみで構成されるサブセット言語を定義し, その言語で記述したモデルを検証技術のモデルへ変換して, 検証を行う手法が提案された [14][15]．それらの手法は, UML 2.x のステートマシンの状態と遷移を非決定性有限状態オートマトンに変換する．しかし, 変換前のモデルが検証したい性質と関連しない情報を含んでいるために複雑な場合や状態が詳細化されて規模が大きい場合には変換後も同じ複雑さや規模になる．よって, 形式検証系が探索する状態空間が極端に巨大化する状態爆発を起こしてしまう．つまり, 現時的なソフトウェア開発におけるモデルに対して, 形式検証技術を適用することは難しい．

3. 本手法の概要

この章では，前節で挙げた技術的課題に対して本手法がとるアプローチについて説明する．次に，アプリケーションドメイン固有の性質を用いたモデル変換手法の概要について説明する．

3.1 本手法のアプローチ

技術的課題に対する解決策として, 何らかの手段により, モデルから検証に関係する部分のみを取り出したり, 状態や遷移をまとめたり省略して状態遷移グラフの複雑さを減らすことによるモデルの抽象化をする必要がある．モデル中に記述されるアプリケーションドメイン固有の役割に注目して, モデルを抽象化することを考える．具体例で説明する．例えば, オンラインショッピングシステムやホテル予約システムなどは「在庫管理アプリケーションドメイン」に属している．従って, それぞれのシステムのモデルにはドメイン固有の役割として, 「在庫」「供給者」「消費者」などが共通して記述されている．また, それらの役割をもったクラス間には「商品や在庫の供給」や「消費」などの関係が記述されている．それらに加えて, 個々のアプリケーションで異なるデータもモデルに記述されている．ここで, これらドメイン固有の役割や関係と関連のある性質のみを検証することを考える．例えば, 「在庫数は常に負にならない」「倉庫やホテル空室の制約から, 在庫数は常にある上限値を越えない」「供給前の在庫数と供給数の和は供給後の在庫数と常に等しい (消費の場合も同じ)」などである．これらの性質の検証には, モデルの全ての情報が必要とは限らない．上記の役割や関係と依存関係を持たない部分はモデルから省略しても検証結果に影響を与えないからである．この省略を本論文では「モデルの抽象化」と定義する．よって，規模や複雑性が大きいモデルに対しても，提案する変換手法によって検証可能になることが期待できる．代償として, あらかじめ定めた特定の性質のみ検証可能となるが, その性質が対象とするアプリケーションにおいて, 重要な性質であれば, この手法は有益となる．アプリケーションドメイン固有の性質検証の実現に当たってはドメインの中での役割を表現する UML プロファイルを定義し, それに沿って記述されたモデルから形式検証が可能な形に変換する．また, アプリケーションドメインに応じて適切な UML プロファイルや変換規則を定義しておくことで，他のドメインに対しても適用可能である．

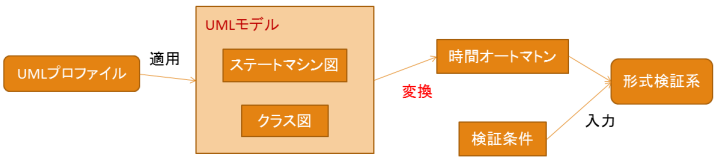

図 1 本手法の構成図

本研究では形式検証技術系として, モデル検査ツールの一つである SPIN モデルチェッカーを用いた．しかし, 本研究の提案手法は SPIN に依存するものでは無く, 別のモデル検査ツールでも適用可能であり汎用性のある手法でもある．また, 基本的な考え方自体は SAT ソルバーや SMT ソルバー等の定理証明系を用いた形式検証技術においても適用可能である．

3.2 本手法の構成

本手法の構成を図 1 に示す．前提として, 対象となるドメインにおいて固有の性質や役割を表す UML プロファイルをあらかじめ定義した．本手法は UML プロファイルを適用してドメイン固有の役割と関係を記述したアプリケーションモデルを入力とする．モデルにはクラス図とステートマシン図が記述されていることを前提とする．入力モデル中のクラス図とステートマシン図に対して, UML プロファイルを適用して記述した役割と関係を元に, それらと関連を持つ部分のみを取り出し, 調べたい性質と直接には関係せず検証結果に影響を与えない部分は捨象し, 抽象化する．この抽象化にもとづいて変換によって得た時間オートマトンを形式検証系に与え, 別途作成した検証条件も与えることで検証結果 (含反例) を得る．

4. 本手法の詳細

この章では, 本手法の詳細について説明する．具体的には, 選択したドメインにおける UML プロファイルの定義とそのプロファイルを適用したモデルの変換・検証手順について説明する．

4.1 UML プロファイル

クラスのステレオタイプ定義について図 2 に示す．ドメインにおいて，<<Consumer>>，<<Provider>>，

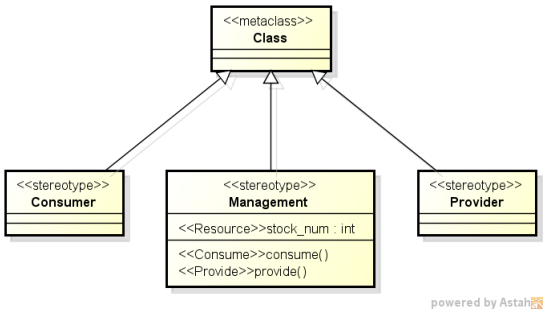

図 2　クラスのステレオタイプ定義

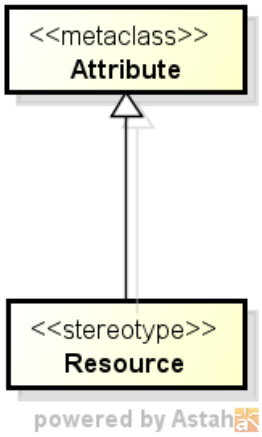

図 3　属性のステレオタイプ定義

<<Management>> のステレオタイプを定義する.

<<Consumer>>
検証に関わる資源を消費するクラスとする.

<<Provider>>
検証に関わる資源を生産するクラスとする.

<<Management>>
生産と消費行動を管理するクラスとする.
在庫の数を示す属性として, <<Resource>> を適用した変数を導入する. 属性ステレオタイプの定義について図 3 に示す. また, 在庫を消費・追加する操作として, それぞれ <<Consume>>, <<Produce>> を適用した操作を道入する. 操作ステレオタイプの定義について図 4 に示す.

定義したステレオタイプの使用例として, 図 5 に示す.

4.2　変換の手順

<<Management>> ステレオタイプが適用されているクラスにおいて, <<Resource>> ステレオタイプが適用されている属性に関連する状態と遷移からなる部分的なステートマシンを抽出する, それにもとづいて時間オートマトンに変換する. 具体的には, 次のような手順で部分的ステートマシン抽出および時間オートマトンへの変換をおこなう.

な お, 前 提 と し て 、<<Consume>> お よ び

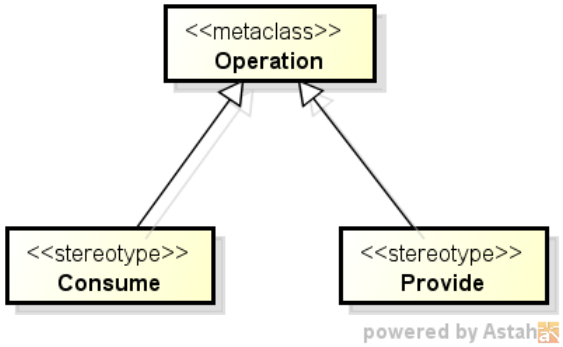

図 4　操作のステレオタイプ定義

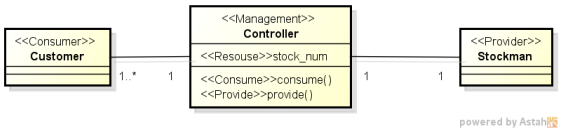

図 5　ステレオタイプの使用例

<<Provide>> ステレオタイプが適用されている操作でのみ <<Resource>> ステレオタイプが適用されている属性変数の値を更新しているとする.

(1) <<Resource>> ステレオタイプが適用されている属性変数を「対象変数」セットに入れる. <<Consume>> および <<Provide>> ステレオタイプが適用されている操作を呼び出しているアクションを持つ状態・遷移に「保持」マークをつける. 以下の手順を新規のマークがつかなくなるまで繰り返す.
(2) 「対象変数」セットに入っている変数を参照・更新している状態・ノード・遷移に「保持」マークをつける.
(3) 「保持」マークがついている状態・ノード・遷移から遷移を逆向きに辿る. その状態・ノードに複数の遷移が入っている場合はそのすべての遷移について逆向きに辿る. 辿っていく途中で, 複数の遷移が出て行く状態・ノードに遭遇した場合は、そこで止める. そこに至るまでのすべての状態・ノード・遷移に「保持」マークをつける.
(4) 3. の状態・ノードから出ている複数の遷移について, それぞれのガード条件に用いられている変数を「対象変数」セットに追加する. それらの遷移に「追跡」マークをつける.
(5) その状態・ノードからさらに逆向きに辿って, 「保持」マークがついている状態・ノード・遷移に到達するまで 3. と 4. を繰り返す.
(6) 「追跡」マークがついている遷移から順方向に辿り, 「保持」マークがついている状態・ノード・遷移に到達したらそこで止める. そこに至るまでのすべての状態・ノード・遷移に, 「追跡」マークがついている遷移も含めて「保持」マークをつける.
(7) 「対象変数」セットに追加された変数について.2 - .5

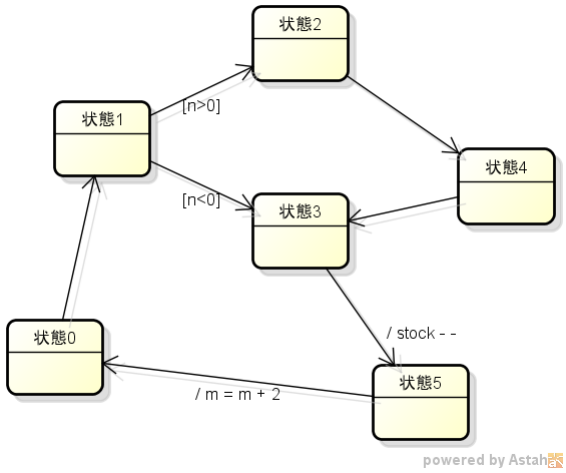

図 6　変換前

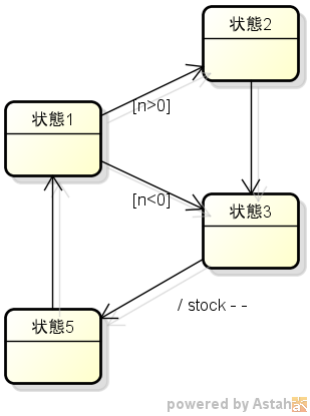

図 7　変換後

を繰り返す.

(8) 「保持」マークがついている状態・ノード・遷移のアクションおよびガード条件について, <<Consume>> および <<Provide>> ステレオタイプが適用されている操作の呼出, および, 「対象変数」セットに入っている変数の参照・更新以外の式や命令文は削除する.

(9) 8. の結果, トリガーもガード条件も持たない遷移は簡約させる. 具体的には, その遷移の出発/到着状態・ノードを 1 つの状態・ノードに簡約する.

(10) 残ったガード条件について, 条件の判断式を状態による判断式に置換する.

ステートマシンに変換を適用した例を図 6 と図 7 に示す. 対象変数セットには初期状態として stock が入っているものとする.

4.3　オートマトンの生成

検証を行うに当たり, 入力として必要となる在庫の数値を表すオートマトンを生成する. なお, 数値データを検証含むことで, 無限の組み合わせになり検証が不可能になってしまうことを避けるため有限個の組み合わせになるような制約を加える. 具体的には, 次の 2 つの制約を加える.

<<Consumer>> ステレオタイプが適用されたクラスが <<Management>> ステレオタイプが適用されたクラスに対して, 最大で N 回の consume シグナルイベントを送る. <<Provider>> ステレオタイプが適用されたクラスが <<Management>> ステレオタイプが適用されたクラスに対して, 最大で N 回の provide シグナルイベントを送る.

この 2 つの制約に対して, 両方のシグナルイベントの生起順序は独立とする.

この制約内で, 全ての状態を取りうるに, N+3 個の状態をもつオートマトンを生成する. 例として N=3 の場合を図 8 に示す. この場合は Nega, Zero, One, Two, Three, Max の状態をもつオートマトンを生成する. Zero, One, Two, Three は実際の在庫の数を表す状態である. Nega はゼロより少ない状態つまり陥ってはいけない下限を表す状態である. Max は同様に陥ってはいけない上限を表す状態である.

検証の際に, 4.2 および 4.3 で得られたオートマトンを入力とする.

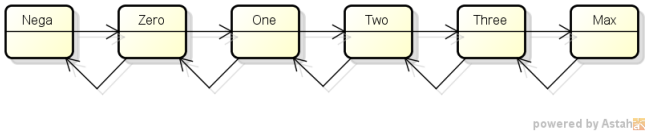

図 8　数値をあらわすオートマトンの例 (N = 3)

4.4　検証条件

在庫管理系アプリケーションドメインにおける, 固有な性質は一例として在庫の数の下限と上限に関連する性質である. 具体的には, 「在庫数は負にならない」および「在庫数は上限値を越えない」である. 本研究では, これらの性質についての検証を行う. 性質を検証系へ与える入力として, LTL による検証条件式の形式で記述すると以下の様になる.

在庫数は常に負にならない ⇒ !(<>(state == nega))

在庫数は上限値を越えない ⇒ !(<>(state == max))

5.　手法の適用事例

本章では本研究における提案手法を実際に適用し検証した結果について述べる. 具体的には, オンラインショッピングアプリケーションについて, UML プロファイルを適用したクラス図とステートマシン図を入力とし, 変換を行い検証する.

5.1　適用事例の概要

適用対象となるアプリケーションの UML クラス図を図

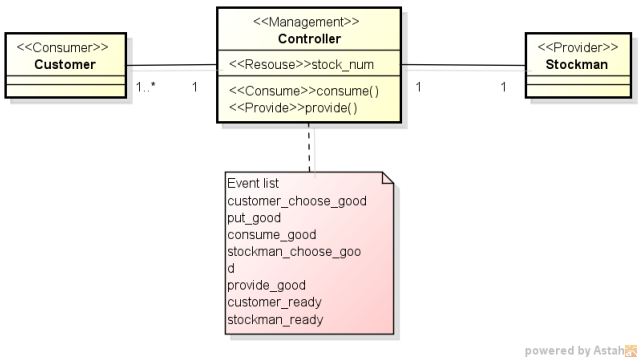

図 9 アプリケーションのクラス図

9, ステートマシン図を図 10 に示す.

<<Consumer>> ステレオタイプが適用された Customer クラス, <<Provider>> ステレオタイプが適用された Stockman クラス, <<Management>> ステレオタイプが適用された Controller クラスから構成されている. Controller クラスは, 属性として <<Resource>> ステレオタイプが適用された在庫数を示す変数 stocknum と 操作として <<Consume>> ステレオタイプが適用された consume() と <<Provide>> ステレオタイプが適用された provide() を有する. Customer クラスおよび Provider クラスは Management クラスに対して, シグナルの送受信を行うことで, 在庫を消費あるいは生産する. 4.3 の制約に対して, N=2 とし高々 2 回のシグナルが生起されるとする.

5.2 変換結果

入力したステートマシン図とクラス図を検証用の時間オートマトンに変換した結果を図 11 に示す.

5.3 検証結果

得られた 2 つのオートマトンを入力として SPIN で検証を行った. なお, 2 つのオートマトンは閉じた系であるため, それら 2 つを駆動するために外部環境を作成し, 検証を行う.

検証条件は, 4.4 にもとづいて, 在庫数の下限と上限に関わる性質から生成した. 具体的には, !(<>(state == nega)) と !(<>(state == nega)) である. SPIN による下限に関わる性質の検証結果 (抜粋) を以下の図 13 示す.

検証結果から, アサーション違反がなく検証条件を満たしていることがわかる. また, 上限に関わる性質の検証についても同様の結果が得られた.

5.4 考察

適用事例では入力としたモデルには誤りがないことを保証した上で手法の適用を行っている. よって, 下限と上限の検証結果として期待されていたのはどちらも条件を満たすことである. つまり, 「在庫数の上限値を越えない」「在庫数は 0 を下回らない」性質を満たすことである. 適用事例の規模・複雑性は大きいとは言えないが, 検証器によって期待された結果が得られたことで, 手法が正しく動作することを確認できた.

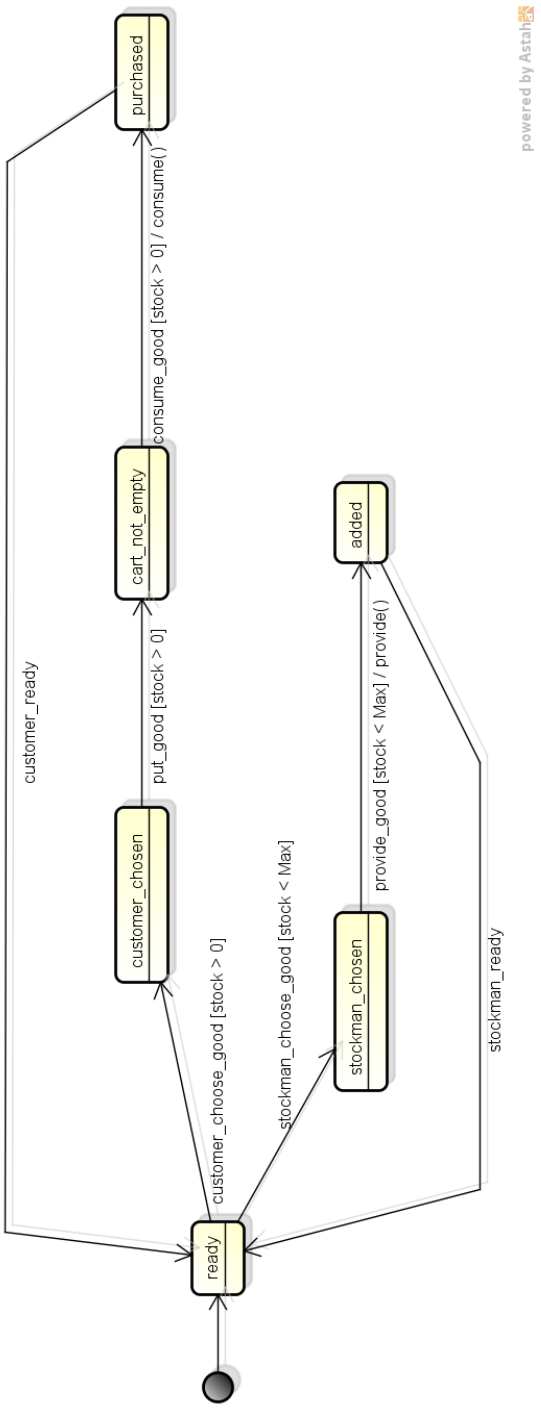

図 10 アプリケーションのステートマシン図

6. 関連研究

文献 [17] では UML/SPT のモデルを時間オートマトンに変換する手法を提案した。UML/SPT は OMG によって標準化された実時間システムを扱うための UML プロファイルである. 本研究と違う点は扱うモデルの複雑さや規模

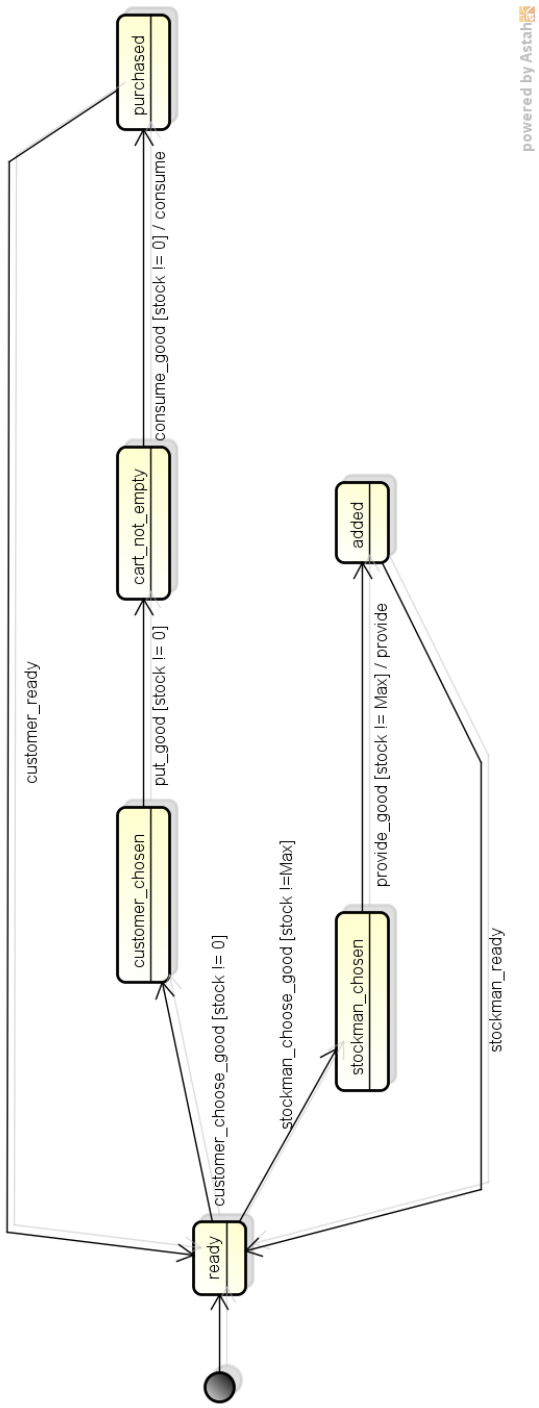

図 11　変換結果 1

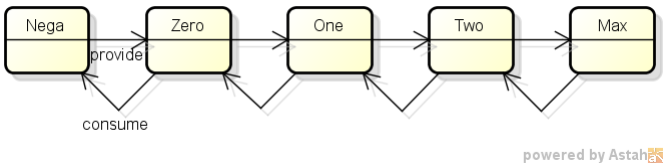

図 12　変換結果 2

に対応していない点である.

文献 [18] では UPPAAL による検証を自動で行う Vooduu というツールを作成した。ツールの入力モデルとしてはス

```
State-vector 48 byte, depth reached 205, errors: 0
     4598 states, stored
     5542 states, matched
    10140 transitions (= stored+matched)
        0 atomic steps
```

図 13　spin による検証結果 (抜粋)

テートマシーン図とシーケンス図を用いて, 時間オートマトンに変換する手法を採っている. モデルの複雑さや規模には対応していない.

文献 [14] では実時間を考慮して HTA を定義し, これを UPPAAL で検証出来るように変換する手法を提案した. HTA とはフラットでは無い, 階層を持ったオートマトンのことである. モデルの複雑さや規模には言及していない.

7. おわりに

大規模化・複雑化するソフトウェア開発に対して, システムをモデルで表現し, 利用する手法が一般化しつつある. 特にモデルの妥当性を検証するために形式検証技術が産業界において注目を集めている. しかし, 開発に用いるモデルは検証したい性質に関わらない情報を含んでおり, 規模や複雑性が大きくなっている. そのため, モデルを形式検証技術に直接適用すると, 状態爆発が生じてしまい検証が行えない. そこで, 本研究ではアプリケーションドメイン固有の性質に注目し. ドメインにおける役割を UML プロファイルで定義し, そのプロファイルを適用したモデルを形式検証可能な形に変換し検証する手法を提案した. 事例に対して手法を適用した結果, 変換の手順が正しく動作することを確認した. 今後は, 複雑な事例に対して, 手法を適用しどの程度抽象化の有効性が認められるか評価したい.

参考文献

[1] UML ”http://uml.omg.org/”
[2] SysML ”http://www.sysml.org/”
[3] SPIN ”http://spinroot.com/spin/whatispin.html”
[4] SMV ”http://www.cs.cmu.edu/ modelcheck/smv.html”
[5] UPPAAL ”http://www.uppaal.org/”
[6] Z ”http://spivey.oriel.ox.ac.uk/mike/zrm/index.html”
[7] VDM ”http://www.vdmportal.org/twiki/bin/view”
[8] SMT yices ソルバー ”http://yices.csl.sri.com/”
[9] HOL-Z ”http://www.brucker.ch/projects/hol-z/”
[10] Coq ”https://coq.inria.fr/”
[11] Selic, B. ”The pragmatics of model-driven development”, Software, IEEE, pp19-25, 2003
[12] Richard J. Anderson, Paul Beame, Steve Burns, William Chan, Francesmary Modugno, David Notkin, Jon D. Reese. ”Model checking large software specifications”, Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering, pp.156-166, 1996
[13] Huafeng Yu, Abdoulaye Gamati, Eric Rutten, Jean-Luc Dekeyser. ”Safe design of high-performance embedded systems in an MDE framework”, Innovations in Systems and Software Engineering, pp.215-222, 2008
[14] Alexandre David, M. Oliver Moller, Wang Yi. ”Formal Verification of UML Statecharts with Real-Time Exten-

sions”, Proceedings of 5th International Conference on Fundamental Approaches to Software Engineering , pp 218-232, 2004

[15] Alexander Knapp, Stephan Merz, Christopher Rauh. “Model Checking Timed UML State Machines and Collaborations”, Proceedings of 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, pp 395-414, 2002

[16] 中島震.”ソフトウェア工学の道具としての形式手法”, NII Technical Report, 2007

[17] Abdelouahed Gherbi, Ferhat Khendek. ”Timed-automata Semantics and Analysis of UML/SPT Models with Concurrency”, Proceedings of 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, pp 412-419, 2007

[18] Karsten Diethers, Michaela Huhn. ”Vooduu: Verification of Object-Oriented Designs Using UPPAAL” , Proceedings of TACAS 2004, pp 139-143, 2004

[19] Matthew B. Dwyer, George S. Avrunin, James C. Corbett. ”Patterns in property specifications for finite-state verification”, Proceedings of the 21st international conference on Software engineering, pp 411-420, 1999

[20] E. M. Clarke , E. A. Emerson , A. P. Sistla. ”Automatic verification of finite-state concurrent systems using temporal logic specifications”, ACM Transactions on Programming Languages and Systems, pp244-263, 1986

[21] Jeannette M. Wing, Mandana Vaziri-Farahani. ”Model Checking Software Systems:A Case Study”, Proceeding SIGSOFT ’95 Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering, pp128-139, 1995

[22] Shin Nakajima. ”Model-Checking Behavioral Specification of BPEL Applications”, Proceedings of the International Workshop on Web Languages and Formal Methods, pp89-105, 2005

[23] Matthew B. Dwyer, George S. Avrunin, James C. Corbett. ”Property specification patterns for finite-state verification”, Proceeding FMSP ’98 Proceedings of the second workshop on Formal methods in software practice, pp7-15, 1998

[24] Dirk Beyer, Georg Dresler, Philipp Wendler. ”Software Verification in the Google App-Engine Cloud”, 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014.

[25] Iwona Grobelna , Michal Grobelny, Marian Adamski. ”Model Checking of UML Activity Diagrams Using a Rule-Based Logical Model”, Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX, Brunow, Poland, June 30-July 4, 2014.