Vol.2013-SLDM-160 No.11
Vol.2013-EMB-28 No.11
2013/3/13

アーキテクチャ指向開発における形式手法の適用に関する考察

張 漢明[1,a)] 野呂 昌満[1,b)] 沢田 篤史[1,c)] 吉田 敦[1,d)] 蜂巣 吉成[1,e)] 横森 励士[1,f)]

概要：本研究の目的はソフトウェアアーキテクチャを中心とした実践的な記述法と検証法を確立することである．振舞い仕様と機能仕様及び詳細化関係に着目して，既存のモデル検査とテスティング技術を適切に適用するための検証モデルを提示する．本稿では単純な自動販売機を事例として，アーキテクチャ段階における仕様記述と検証例を示す．本検証モデルは，ソフトウェア開発者がアーキテクチャを記述及び検証するさいの実践的な指針となることを目指す．

A Discussion of Applying Formal Methods to Architecture Oriented Development

Han-Myung Chang[1,a)] Masami NORO[1,b)] Atsushi SAWADA[1,c)] Atsushi Yoshida[1,d)]
Yoshinari Hachisu[1,e)] Reishi Yokomori[1,f)]

Abstract: We aim to establish practical specification and verification methods for architecture oriented software development. We pay attention to behavioral specifications, functional specifications and refinement relations, and propose verification models in order to apply existing model checking and testing technologies to the development systematically. In this paper, architectural specifications and verifications of a simple vending machine are presented as a case study. The verification models will be used as practical guidelines for architectural specifications and verifications.

1. はじめに

マルチコアプロセッサの出現や分散システムの普及に伴い，並行プログラミング技術の重要性が注目されている．一般的に，並行システム記述をテスティングやレビューで検証することは不可能である．また，開発者が安全のために防衛的なプログラムを作成したり，詳細化により元の構造が崩れることで，プログラム全体の見通しが悪くなる．

システム開発の上流工程で並行システムの設計を検証するためのモデル検査の有用性が報告されている [2], [8]．並行システムの状態を網羅的に検査することにより，予期せぬ振舞いや状態を検出することができる．モデル検査では状態爆発の問題があるので，モデル検査とテスティングを用いて効率よく検証する方法が求められる．

本研究の目的は，ソフトウェアアーキテクチャを中心とした実践的な記述法と検証法を確立することである．仕様，アーキテクチャ，設計，コード間のソフトウェア文書の一貫性を保持するために，形式言語を用いた検証法を提示することを目指す．本稿では，仕様とアーキテクチャ記述について検討する．

本研究では，振舞い仕様と機能仕様及び詳細化関係に着目して，既存のモデル検査とテスティング技術を適切に適用するための検証モデルを提示する．振舞い仕様は，システムの状態に依存しない抽象的な仕様と捉える．一方，機能仕様は入力，状態変化，出力の関係を定義するが，ここではシステムが保持する状態不変条件を機能仕様の基準として捉える．

[1] 南山大学情報理工学部ソフトウェア工学科
[a)] chang@nanzan-u.ac.jp
[b)] yoshie@nanzan-u.ac.jp
[c)] sawada@nanzan-u.ac.jp
[d)] atsu@nanzan-u.ac.jp
[e)] hachisu@nanzan-u.ac.jp
[f)] yokomori@nanzan-u.ac.jp

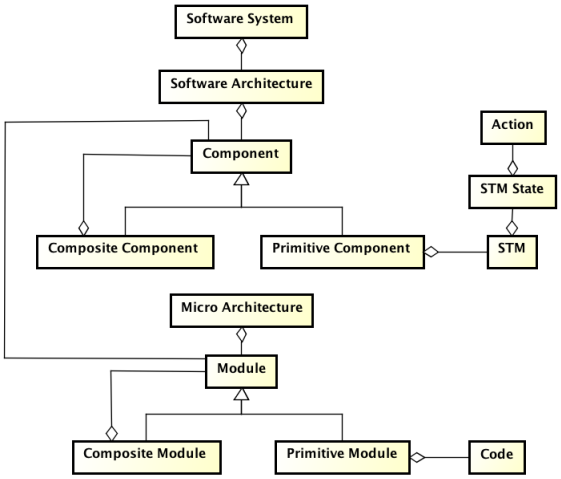

図 1 ソフトウェアアーキテクチャ
Fig. 1 Structure of Software Architecture

本稿では，アーキテクチャを検証するための検証モデルを提示し，単純な自動販売機を事例として，アーキテクチャ段階における仕様記述と検証例を示す．振舞い仕様はプロセス代数 CSP[9] を用いて記述し，CSP の代表的なモデル検査器 FDR を用いて検証する．状態不変条件は，状態の関係を定義するのに適切な Z 言語 [11] を用いて記述する．

2. 基本的なアイデア

本研究で想定するソフトウェアアーキテクチャの構造と記述及び提案する検証モデルについて説明する．

2.1 ソフトウェアアーキテクチャ

プログラムもしくはコンピュータシステムのソフトウェアアーキテクチャは，システムがどのように振舞うかを理解するためのシステムの記述であると定義されている [10]．本研究では，コンポーネントベースソフトウェア開発 [7] を想定し，システムは並行に動作する状態遷移機械の集合としてモデル化されるものとする．組込みシステムに代表される並行システムでは，システムの振舞いは状態遷移モデルで記述されることが一般的である [12]．

本研究で想定するソフトウェアアーキテクチャの構造を図 1 に示す．ソフトウェアアーキテクチャは複合コンポーネントとして構成され，コンポーネントの振舞いは状態遷移機械として表される．アーキテクチャ段階の構成物であるコンポーネントは，ソースコードを表すモジュールとして実現される．モジュールの構成を表すマイクロアーキテクチャはプログラムの設計に利用されているデザインパターン [5] に対応する．本稿では，アーキテクチャ段階の記述であるコンポーネントの記述と検証について議論する．

2.2 検証モデル

提案する検証モデルの概要を図 2 に示し，検証モデルを

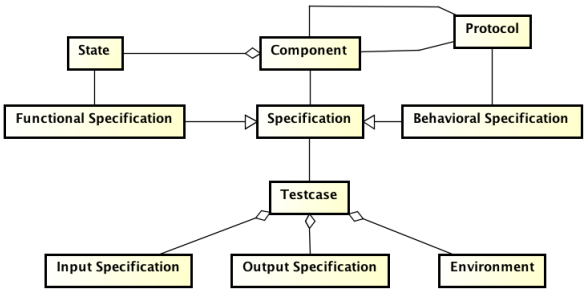

図 2 検証モデル
Fig. 2 Verification model

「振舞い仕様と機能仕様」，「抽象モデルと具象モデル」，「詳細化関係」の観点から説明する．

2.2.1 テストケース

テストの項目を表すテストケースは入力仕様，出力仕様，環境から構成される．テストケースは，コンポーネントの記述が仕様を満たすことを検証するための記述である．環境はソフトウェアシステムが接する外界を表す．組込みシステムでは，環境はハードウェアつまりデバイスドライバに対応する．テストケースは検証の対象とするコンポーネントにおいて，環境からの入力に対して想定する出力を記述する．

2.2.2 振舞い仕様と機能仕様

コンポーネントの仕様には「振舞い仕様」と「機能仕様」がある．振舞い仕様は，対象システムの事象をイベントとしてモデル化し，イベントの実行順序の関係をプロトコルとして定義する．機能仕様は，コンポーネントが保持する「状態」に対して，その状態変化を関係と捉えて定義する．振舞い仕様では「状態」に依存しない振舞いを定義することにより，抽象度が高い記述を提供する．

本研究では，アーキテクチャ段階で記述するイベントを，コンポーネント間の送受信とする．システムが満たすべき振舞いの性質やデッドロックが起こらないという性質をアーキテクチャの段階で保証することにより，プログラミングからの戻り作業が減少することを期待している．ソースコードのモジュールを設計する作業では，ソースコードに対する振舞い仕様が検証条件となる．

振舞い仕様のテストケースは，環境からの入力イベントの順序関係に対して，想定する出力のイベントの順序関係を定義する．機能仕様のテストケースは，機能の実行前の状態と入力と，実行後の状態と出力の関係を定義する．機能仕様では，システムが常に満たすべき「状態不変条件」の記述が最も重要である．

2.2.3 抽象モデルと具象モデル

ソフトウェアアーキテクチャの定義は図 3 に示されているように，対象システムの本質を表す「抽象モデル」と実体を表す「具象モデル」に分離して考える．抽象モデルで

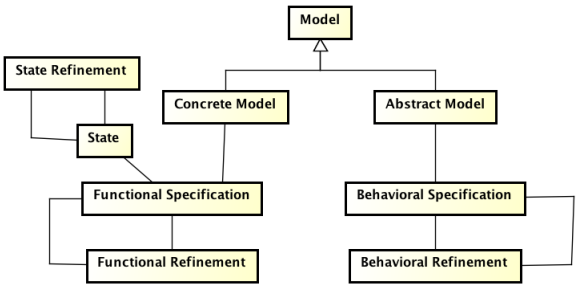

図 3 抽象モデルと具象モデル
Fig. 3 Abstract model and concrete model

は，具体的な実体や環境に依存しないモデルを構築する．具象モデルでは，環境や実体に対応したモデルを構築する．例えば，MVC アーキテクチャにおけるモデルを抽象モデル，ビューとコントローラを具象モデルと捉えることができる．

抽象モデルの仕様は，状態に依存しない振舞い仕様として定義する．具象モデルの状態や詳細な情報はイベントの付加情報として反映される．抽象モデル間の振舞いではこれらの付加情報は隠蔽されて，状態，環境，具象モデルの詳細化や変更に影響を受けないようにする．

2.2.4 詳細化関係

仕様の詳細化 (Refinement) 関係は図 3 に示されているように，「振舞いの詳細化 (Behavioral Refinement)」及び「状態の詳細化 (State Refinement)」と状態に関連する「機能の詳細化 (Functional Refinement)」が考えられる．

振舞いについては，「イベントの詳細化」と「際どいイベントの対応」がある．イベントの詳細化は，1 つのイベントでモデル化したものを，複数のイベントに置き換える．例えば，自動販売機で商品を排出することをモデル化する場合に，output イベントで表わしたものを，output_request（排出指示）と outout_ok（排出完了）で表すことをイベントの詳細化とする．

際どいイベントの詳細化は，イベントの際どい順番を考慮した振舞いに対応する．例えば，自動販売機で購入ボタンと返却レバーを同時に発生させた場合の振舞いに対応することがある．通常はこのような対応を行うと，状態遷移機械の状態数が増え，状態遷移機械の理解が困難なものとなる．そこで，際どい振舞いの対応部分を 1 つのコンポーネントに分離する．

機能の詳細化は状態の詳細化に対応して行われる．例えば，集合として表された購入ボタンに，button_id を割り当てたり，商品の価格と対応づけたりすることを状態の詳細化とする．機能の詳細化は状態の詳細化に対応して定義するが，ここでは状態が満たすべき関係を表す状態不変条件に着目する．

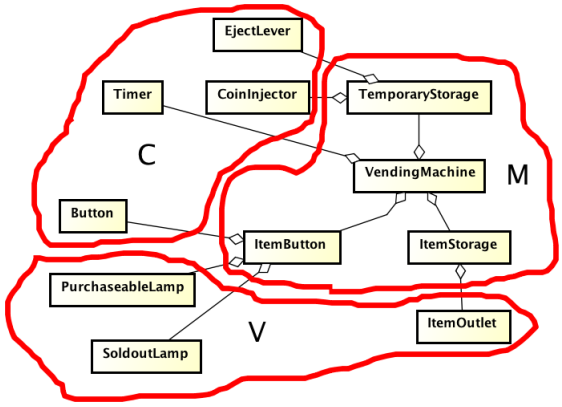

図 4 自動販売機のソフトウェアアーキテクチャ
Fig. 4 Software architecture of vending machine

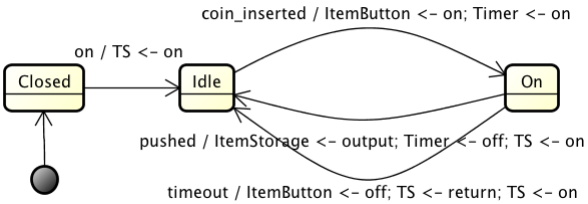

図 5 VendingMachine の振舞い
Fig. 5 Behavior of VendingMachine

2.2.5 仕様記述と検証ツール

振舞い仕様は，状態に依存せずにイベントの送受信としてモデル化するので，プロセス代数 CSP[9] を用いるのが自然である．振舞い仕様の検証は CSP の代表的なモデル検査器である FDR[4] を用いる．

機能仕様は，モデル検査とテスティングを用いて検証する．モデル検査では少数のインスタンスを用いて網羅的に無限の振舞いを検証する．テスティングでは状態不変条件に基づいて入力仕様，出力仕様，環境のテストケースを構成する．状態不変条件は状態の関係を記述するのに適した Z 言語 [11] を用いて記述する．

3. 事例：自動販売機システム

自動販売機システムを事例として，ソフトウェアアーキテクチャの記述と検証例を示す．

3.1 ソフトウェアアーキテクチャ

自動販売機システムのソフトウェアアーキテクチャを図 4 に示す．自動販売機を MVC（モデル，ビュー，コントローラ）アーキテクチャとしてモデル化した．モデルは自動販売機の本質を抽象化した VendingMachine, TemporaryStorage, ItemButton, ItemStorage で構成されている．

3.2 抽象モデルの仕様と検証

抽象モデルとして MVC におけるモデルの記述と検証

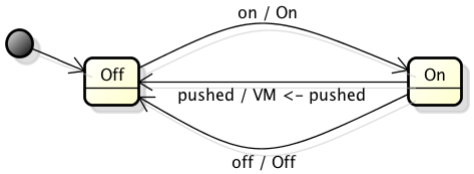

図 6 ItemButton の振舞い
Fig. 6 Behavior of ItemButton

例を示す．図 5 は自動販売機の基本的な振舞いを定義した VendingMachine の状態遷移機械である．自動販売機はコインが挿入される (coin_inserted) と ItemButton と Timer に on イベントを送信して，ボタンとタイマを有効にする．

ボタンが押される (pushed) と ItemStorage に output イベントを送信して商品を排出し，Timer に off イベントを送信してタイマを停止する．逆に Timer がタイムアップする (timeout) と ItemButton に off イベントを送信してボタンを無効にし，TS(TempraryStorage) に return イベントを送信してコインを返却する．ItemButton の振舞いを図 6 に示す．Timer の振舞いも ItemButton と同様である．

商品購入の振舞い仕様の検証を以下に示す．入力仕様を INPUT として定義する．

```
INPUT = Repeat(CoinInjected; ButtonPushed)
CoinInjected =
      rcv.TS.on -> snd.TS.injected -> SKIP
ButtonPushed = rcv.ItemButton.on ->
             snd.ItemButton.pushed -> SKIP
```

出力仕様を OUTPUT として定義する．

```
OUTPUT = CoinInjected -> ButtonPushed
                  -> ItemOutput -> OUTPUT
ItemOutput = rcv.ItemStorage.output -> SKIP
```

モデル検査は以下の詳細化関係を検査する．

```
OUTPUT [F= Input [|SYNC|] Components
```

[F= は CSP の失敗モデルにおける詳細化関係を表している．SYNC は 入力とコンポーネント間で同期するイベントで

```
SYNC = {rcv.TS.on, rcv.ItemButton.on}
```

である．

3.3 具象モデル

具象モデルでは，コントローラのボタンデバイスに対応する環境を含めた振舞い仕様の検証を行う．環境としてボタンのデバイスを定義する．

```
ButtonDevice = rcv.Button.on ->
    (snd.Button.pushed -> ButtonDevice []
     rcv.Button.off ->ButtonDevice)
```

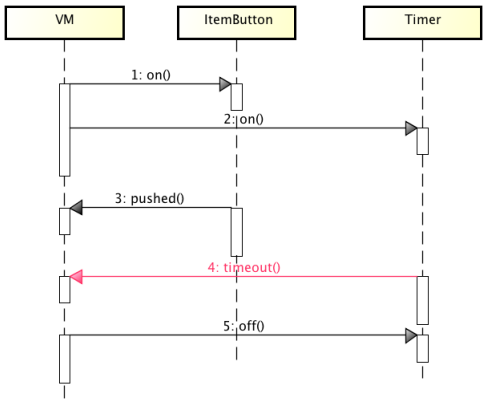

図 7 イベントの競合
Fig. 7 Event competition

ボタンデバイスは Button が有効 (on) になった後で，ボタンが押される (pushed) か無効 (off) になる．入力仕様を INPUT として定義する．

```
INPUT = Repeat(CoinInjected; ButtonPushed)
CoinInjected =
    snd.CoinInjector.coin_inserted -> SKIP
ButtonPushed = snd.button.pushed -> SKIP
```

モデル検査は環境を含めた以下の詳細化関係を検査する．

```
OUTPUT [F= Input [|SYNC|]
     (Environment [|SYNC_ENV|] Components)
```

SYNC_ENV は 環境とコンポーネント間で同期するイベントで

```
SYNC_ENV = {rcv.Button.on, rcv.Button.off}
```

である．

3.4 振舞いの詳細化

振舞いの詳細化を具体例を用いて説明する．商品購入の入力仕様を，以下のようにコインの挿入とボタン押下が並行に動作するものとする．

```
INPUT = Repeat(CoinInjected) |||
                    Repeat(ButtonPushed)
```

コインはコインが挿入できる場合に挿入し，ボタンはボタンが押下できる場合に押すことを表している．このとき出力仕様として，商品が排出されるか，コインが返却されることが想定される．

```
OUTPUT = Repeat(ItemOutput [] CoinReturned)
```

しかしながら，この検査は失敗する．

図 5 における基本的な振舞いの On の状態で，図 7 のように，ボタンが押された (pushed) 時にタイマを停止させる (off) 前に timeout イベントを受信する場合がある．図 5 ではこの順番でイベントが起こることを想定していない．このような状況を「イベント競合」と呼ぶことにする．

イベント競合とは，ある状態で複数のイベントを待って

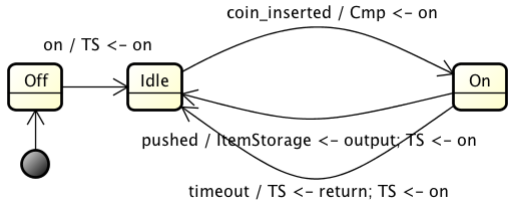

図 8 VendingMachine-alt の振舞い
Fig. 8 Behavior of VendingMachine-alt

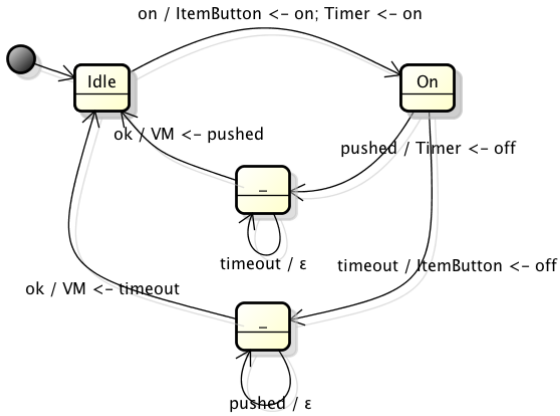

図 9 競合コンポーネント (Cmp)
Fig. 9 Competition component(Cmp)

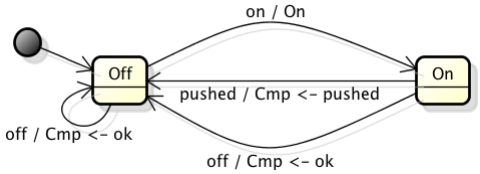

図 10 ItemButton-alt の振舞い
Fig. 10 Behavior of ItemButton-alt

いる時に，複数のイベントを受信する状況をいう．上記のイベント競合に対応した状態遷移機械を図 8 に示す．イベント競合に対応する部分は図 9 の状態遷移機械 (Cmp) が行っている．Cmp は ItemButton もしくは Timer に off イベントを送信して，競合が起こったかどうかを調べている．ItemButton はこの off イベントに対応するために図 10 の状態遷移機械となる．この記述例では先に起こったイベントを優先させている．

3.5 状態の詳細化

状態の詳細化として「ボタンの詳細化」と「購入ランプ」を例として説明する．

3.5.1 ボタンの詳細化

これまでの抽象モデルでは，ボタンの詳細を取捨したモデルで自動販売機の振舞いを検討した．ここで，ボタンが複数ある場合の検証について考える．

まず，ボタンの集合を定義する．

$$[BUTTON]$$

$BUTTON$ は Z 言語 [11] で集合を表すが，その具体的な要素は言及していない．これは抽象的な仕様を書く場合に有効な手段である．

ボタン押下のイベント pushed を以下のように定義する．

```
channel pushed:BUTTON
```

pushed はボタンの情報を付加した複合イベントとなる．商品購入の振舞い仕様を以下に示す．

```
REPEAT([] b:BUTTON @ snd.button.pushd.b ->
                    rcv.itemStorage.output.b -> SKIP)
```

上記は，ボタンで選択されたボタン (pushed.b) の商品が排出 (output.b) することを表している．モデル検査で検証するには具体的なデータが必要となるが，ボタンの具体的なデータを用いずに定義していることによりモデル検査とテスティングの両方の仕様として使うことができる．

3.5.2 購入ランプ

購入ランプを例として状態不変条件による機能の仕様を示す．購入ランプに関連する自動販売機の状態を Z 言語を用いて定義する．

$$CURRENCY == \mathbb{N}$$
$$OnOff ::= on \mid off$$

VendingMachine

$$buttons : \mathbb{P}\, BUTTON$$
$$input : CURRENCY$$
$$price : BUTTON \pfun CURRENCY$$
$$purchasableLamp : BUTTON \pfun OnOff$$

$$buttons = \mathrm{dom}\, price$$
$$buttons = \mathrm{dom}\, purchasableLamp$$

VendingMachine はボタン *buttons*，入力金額 *input*，価格 *price*，購入ランプ *purchasableLamp* の状態を持つことを表している．購入ボタンの定義は以下のようになる．

PurchasableLamp

$$VendingMachine$$

$$\forall\, b : buttons \bullet price(b) \leq input \Leftrightarrow purchasableLamp(b) = on$$

PurchasableLamp は，価格が入力金額以下のランプが *on* になることを表している．これは，複数ボタンの仕様と同様に，具体的なデータに言及せずに購入ランプの機能の本質を定義している．

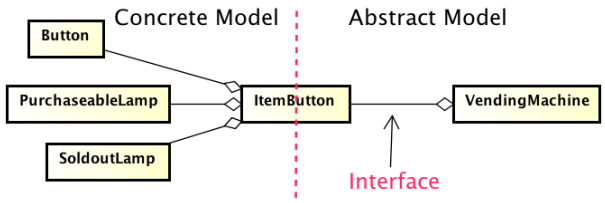

図 11 検証の分割
Fig. 11 Deviding Verifications

4. 考察

「再利用」と「検証の分割」について考察する．

4.1 再利用

コンポーネントベースのソフトウェア開発において，振舞いの詳細化，テストケース，ポリシーについて検討する．

振舞いの詳細化

第 3.4 節のイベント競合の例では，イベント競合に対応する部分をコンポーネントとしてモジュール化した．イベント競合は非同期通信の並行システムにおいて常に考慮する必要がある．イベント競合に対して，先行イベント優先やイベントの優先度などを用いたモジュールをライブラリの部品として提供するこができる．

テストケース

事例で示した振舞い仕様や機能仕様は，抽象度の違うテストケースの作成が考えられる．テストケースの仕様として，内部構造を取捨した抽象的な仕様を提供することにより，仕様の共有化を図り，具体的なテストケースを状態と詳細化関係から生成する．

ポリシ

図 5 は自動販売機の販売ポリシーを，基本的な振舞いとして記述しているとみなすことができる．例えば，ボタンを押してからコインを挿入する自動販売機を考える．これは販売ポリシ `VendingMachine` を作成するだけで構築することができることが望ましい．そのためには，ドメイン分析を行い可変部を特定することが重要である．

4.2 検証の分割

実用的な大規模なシステムを構築するには，システムの分割が必要である．検証の観点から分割について検討する．本稿では抽象モデルと具象モデルでモデルを分けて考えた．抽象モデルは具象モデルに依存しないモデルを提供する．抽象モデル間の振舞いをインターフェースと呼ぶことにする．

この検証の分割を図 11 に示す．具象モデルと抽象モデルの境界にあるのが `ItemButton` である．`ItemButton` と `VendingMachine` の間のプロトコルがインターフェースである．このインターフェースを以下に示す．

```
Interface = Button.on -> Timer.on ->
  (pushed -> Interface [] timeout -> Interface)
```

従って，抽象モデルの振舞いと具象モデルの振舞いが `Interface` と同一になることを検証することにより検証の分割が可能となる．

5. おわりに

本稿では，コンポーネントベースのソフトウェア開発において，アーキテクチャ段階での記述と検証を行うための検証モデルを提示して，自動販売機システムを事例として振舞い仕様と検証例を示した．

関連研究として，組込みシステムのためのアプリケーションフレームワーク生成系の VARTAF[6] がある．ここでは状態遷移機械に基づいた検証の支援を行っているが，詳細化関係を扱っていない．また，モデル検査で検証した性質をソースコードにおいて保証する整合テストの研究 [1] が行われている．我々はモデル検査とソースコードの共通仕様からそれぞれのテストケースを作成することを試みている．

謝辞 本研究の一部は，科学研究費補助金（基盤研究（C）21500042, 22500036, 22500037，24500049，基盤研究（S）24220001）および 2012 年度南山大学パッヘ奨励金 1-A-2 の助成を受けて実施した．

参考文献

[1] 青木利晃, T. T. M., Nguyen: モデル検査による設計検証と整合テスト, 情報処理学会研究報告, EMB, 組込みシステム 2009-EMB-14(4), pp. 1–8 (2009).
[2] Ben-Ari, M.: Principles of Concurrent and Distributed Programming 2nd edition, Addison-Wesley (2006).
[3] 張漢明, 野呂昌満, 沢田篤史, 蜂巣吉成, 吉田 敦: モデル検査を用いた振舞い検証の実用化技術に関する考察, FOSE2010, 近代科学社 (2010), pp. 107–112.
[4] FDR2 Manual, Formal Systems Limited.
[5] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: 本位田真一, 吉田和樹 (監訳), オブジェクト指向における再利用のためのデザインパターン (改訂版), ソフトウェアクリエイティブ (1999).
[6] P.A. Hsiung and C.W. Lin and C.H. Tseng and T.V. Lee and J.M. Fu and W.B. See: VARTAF: An Application Framework for the Design and Verification of Embedded Real-Time Software, IEEE Trans. on SE, 30, 10, pp. 656–674 (2004).
[7] Ning, G.: A Conponent-Based Software Development Model, COMPSAC'96, pp. 389–394 (1996)
[8] 中島震：モデル検査法のソフトウェアデザイン検証への応用, コンピュータソフトウェア, Vol.23, No.2 (2006), pp. 72–86.
[9] Roscoe, A.W.: Understanding Concurrent Systems, Springer (2010).
[10] Software Architecture, http://www.sei.cmu.edu/architecture/.
[11] Spivey, M.: The Z Notation, Prenntice Hall (1992).
[12] 沢田篤史, 平山雅之 (編):組込みソフトウェア開発技術, CQ 出版社 (2011).