Vol.2010-EMB-18 No.6
2010/8/9

階層分割に基づく組込みソフトウェアの振舞い検証の支援について

張　漢明[†1]　野呂昌満[†1]　沢田篤史[†1]
蜂巣吉成[†1]　吉田敦[†1]

モデル検査を用いた振舞い検証における解決すべき課題として，厳密な振舞いの仕様の作成と状態爆発の回避があげられる．本稿では，階層的に分割されたシステムに対する網羅性に着目した検証のモデルを提案する．プロセス代数 CSP のモデル検査器 FDR を用いた検証方法と適用事例を提示し，振舞い検証の支援技術について議論する．

A discussion on the behavioral verification of embedded software based on hierarchical models

Han-Myung CHANG,[†1] Masami NORO,[†1]
Atsushi SAWADA,[†1] Yoshinari HACHISU[†1]
and Atsushi YOSHIDA[†1]

One of the most important issues in behavioral verification using model checking is how to specify rigorous behavioral specification and how to resolve the state explosion problem. We propose a verification model focusing on the exhaustiveness for hierarchical model. We show verification methods with FDR, which is a model checking tool for CSP, and case studies, and discuss supporting techniques for behavioral verification.

†1 南山大学
Nanzan University

1. はじめに

近年，組込みシステムの大規模化と複雑化に伴い，開発現場への本格的なソフトウェア開発技術の適用が求められている[1)]．イベント駆動に基づいた並行システムのソフトウェア開発において，組み合わせ的に発生するイベントを正しく制御することは困難な作業である．ソフトウェア開発の大部分は，設計時に想定していなかった，特に異常系のイベントの処理に費やされていると言っても過言ではない．システムの状態を網羅的に探索してシステムの性質を検証するモデル検査技術が注目されている．モデル検査を用いた振舞い検証を実用化するさいの障壁として，振舞い仕様を厳密に記述する困難さと状態爆発の問題がある．

本研究の目的は，モデル検査を用いた振舞い検証における仕様記述の支援と状態爆発を回避する方法を提示することである．モデル検査技術を実際のソフトウェア開発の現場で適用するには，検証の手順と検証の基準を明示することが必要である．モデル検査は，システムの振舞いの記述に対して満たすべき性質を検証する方法を提示しているが，どのような検証をどこまで行えば良いかは示していない．また，状態爆発の問題を解決するために分割統治法を用いるには，分割されたモジュールを合成するさいに不整合が生じないことを検証する必要がある．

本研究の方針は，階層モデルで構成されるシステムのアーキテクチャ記述に対して網羅性の基準を明示して，網羅性を検証するための方法を提示することである．本研究では，上位コンポーネントと下位コンポーネント間のインターフェースに着目した．インターフェースは上位コンポーネントと下位コンポーネント間の振舞いを記述したものである．網羅性は，下位コンポーネントが想定する全ての振舞いに対する上位コンポーネントへの振舞いが，インターフェースと一致することとする．網羅性は，コンポーネントを合成するさいに，下位コンポーネントと関連する下位のコンポーネント群の代わりに，インターフェースを用いることを可能にする．

本稿では，インターフェースの網羅性に着目した検証のモデルを提案し，プロセス代数 CSP[2)] のモデル検査器 FDR[3)] を用いた検証方法と適用事例を提示する．検証のモデルはモデル検査の言語に依存しない形で網羅性の概念を定義する．コンポーネントは状態遷移機械として表現され，アクションにはイベントの送信が記述される．状態遷移機械を CSP で表現するためのライブラリを提示し，網羅性を保証するための FDR による検証方法を示す．インターフェースの網羅性が検証の手順と検証の基準となり，アーキテクチャ記述の信頼性向上に寄与する．

Vol.2010-EMB-18 No.6
2010/8/9

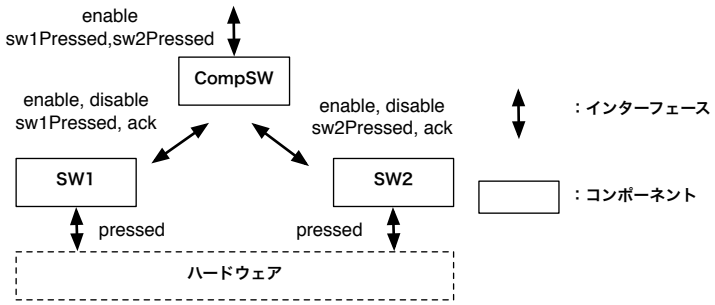

図 1 複合スイッチの階層構造
Fig. 1 Structure of a composite switch

2. 基本的なアイデア

本研究では，コンポーネントに基づいたソフトウェアアーキテクチャ[4] 記述に対するコンポーネントの振舞い検証を対象とする．アーキテクチャ記述におけるコンポーネントは状態遷移機械として表現され，アクションにはイベントの送信が記述される．複数のコンポーネントで構成される制御の見通しを良くするために，コンポーネントは階層モデルで構成されているものとする．

ここでは，2 つのスイッチで構成された複合スイッチを事例として「振舞い仕様」「インターフェース」および「網羅性」の観点から検証の概念を説明する．複合スイッチの構成を図 1 に示す．複合スイッチ（CompSW）は下位コンポーネントのスイッチ 1（SW1）およびスイッチ 2（SW2）から構成されている．CompSW の機能は SW1 もしくは SW2 からスイッチが押された情報を上位コンポーネントに伝えることである．ただし，SW1 と SW2 が同時に押された場合には SW1 を優先して通知するものとする．

2.1 振舞い仕様

コンポーネントが満たすべき振舞いに関する性質を「振舞い仕様」と呼ぶ．振舞い仕様は，コンポーネントへのイベントの入力に対するイベントの入出力として表現する．図 1 の例では「SW1 を有効（enable）にしてハードウェアから SW1 が押された（pressed）という通知があると，CompSW に SW1 が押された（sw1Pressed）を通知する」という記述や「CompSW を有効（enable）にして，CompSW に SW1 が押された（sw1Pressed）という通知があると，SW1 が押された（sw1Pressed）が上位コンポーネントに通知される」という記述が振舞い仕様である．

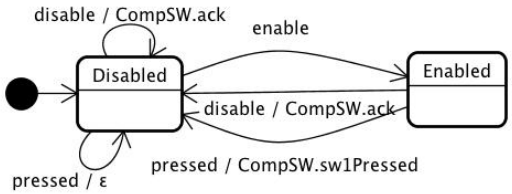

図 2 スイッチ 1（SW1）の状態遷移図
Fig. 2 The state transition diagram for SW1

2.2 インターフェース

振舞い仕様の入力に対するイベントの入出力が，上位コンポーネントとの間の振舞いを表現している場合，その振舞いをインターフェースの振舞いと呼ぶ．図 1 の例では，SW1 のインタフェースの振舞いは SW1 とその上位コンポーネントである CompSW との間で送受信されるイベント（enable, disable, sw1Pressed, ack）を用いて記述される．ここで SW1 の状態遷移図は図 2 とする．

SW1 はスイッチが有効の状態（Enabled）のときハードウェアから押されたという通知（pressed）は CompSW に通知（sw1Pressed）され，無効の状態（Disabled）のときは押されたという通知は破棄される．また，SW1 は CompSW からのスイッチの無効指示（disable）に対して応答（ack）を返している．これは，CompSW が disable を送信したと同時に SW1 が sw1Pressed を送信したさいに，この sw1Pressed を検知するために ack を返している．CompSW の状態遷移図を図 3 に示す．図 3 における WaitAck が SW1 と SW2 が同時に押された状態である．

SW1 のインターフェースは，コンポーネント間の通信方式が非同期通信であれば，Disabled の状態で enable を受信してから

(1) sw1Pressed を CompSW に送信
(2) disable を受信して ack を送信
(3) disable を受信して sw1Pressed および ack を送信
(4) sw1Pressed を送信した後に disable を受信して ack を送信

の振舞いが考えられる．

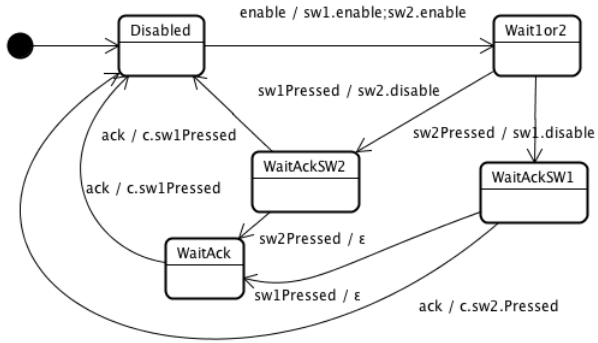

図 3 複合スイッチ（CompSW）の状態遷移図
Fig. 3 The state transition diagram for CompSW

2.3 網 羅 性

網羅性は，下位コンポーネントが想定する全ての振舞いに対する上位コンポーネントへの振舞いが，インターフェースと一致することである．SW1 の例では，CompSW とハードウェアからの想定される入出力の組み合わせを考慮する必要がある．CompSW から SW1 には，enable および enable の後に disable が送信され，ハードウェアからは常に pressed が送信される可能性がある．前述の SW1 のインターフェースの記述が，想定される全ての組み合わせに対する振舞いを表現している場合，インターフェースは網羅性を満たしていると言う．

インターフェースの網羅性は，コンポーネントを合成するさいに，下位コンポーネントと関連する下位のコンポーネント群の代わりに，インターフェースを用いることを可能にする．CompSW のインターフェースは，CompSW を有効（enable）にした後で SW1 が押された（sw1Pressed）もしくは SW2 が押された（sw2Pressed）が上位コンポーネントに通知される．インターフェースが網羅性を満たしていれば，上位コンポーネントは CompSW の振舞いとしてこのインタフェースを用いることができる．

インターフェースをコンポーネントの振舞いとして用いるには，網羅性を形式的に検証する必要がある．インターフェースの網羅性を検証する基本的な考え方は，上位コンポーネントおよび下位コンポーネントのインターフェースの振舞いと，対象コンポーネントの振舞いを合成したものが，インタフェースを満足することを示すことである．第 3 章で検証の概念を提示し，CSP による検証の方法を第 4 章で説明する．

3. 振舞い検証のモデル

振舞い検証のモデルを Z 記法[5] を用いて定義する．ここでは，振舞いの表記法は規定せずに検証の概念モデルを提示する．

3.1 振 舞 い

振舞いはイベントの送受信として表現される．

$[EVENT, BEHAVIOR]$

$EVENT$ はイベントの集合を表し $BEHAVIOR$ は振舞いの表現を表す．イベントと振舞いの具体的な記述はここでは規定しない．

$$\begin{array}{|l} \textit{Behavior} \\ \hline events : \mathbb{P}\ EVENT;\ behavior : BEHAVIOR \\ \hline \end{array}$$

$Behavior$ はイベントの集合（$events$）と振舞い（$behavior$）から構成されている．$behavior$ は $events$ で表現された振舞いであるとする．

振舞いを合成する方法として $concurrent$ と $choice$ がある．

$$concurrent, choice : \mathbb{P}\ BEHAVIOR \longrightarrow BEHAVIOR$$

$concurrent$ と $choice$ は振舞いの集合から振舞いへの関数で，$concurrent$ は並行合成，$choice$ は選択合成を表す．

振舞いの関係として $satisfy\_$ がある．

$$satisfy\_ : BEHAVIOR \leftrightarrow BEHAVIOR$$

$(A, B) \in satisfy\_$ は A が B を満足することを表す．$concurrent$，$choice$，$satisfy\_$ は具体的な言語の基で定義される．

3.2 振舞い仕様

振舞い仕様（$Specification$）は入力（$input$）と入力に対する入出力（$inputOutput$）から構成される．

$$\begin{array}{|l} \textit{Specification} \\ \hline input, inputOutput : Behavior \\ \hline input.events = inputOutput.events \\ \hline \end{array}$$

$input$ と $inputOutput$ は振舞い（$Behavior$）を表し，それぞれの対象とするイベントは同

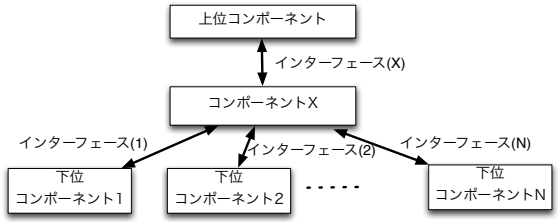

図 4 階層モデル
Fig. 4 Hierarchical model

じである．

振舞いが振舞い仕様を満足することを関係 *satisfy* で表現する．

$$satisfy : BEHAVIOR \leftrightarrow Specification$$

$$\forall target : BEHAVIOR; spec : Specification \bullet (target, spec) \in satisfy \Leftrightarrow (concurrent(\{spec.input.behavior, target\}), spec.inputOutput.behavior) \in satisfy\_$$

$(target, spec) \in satisfy$ は振舞い *target* が振舞い仕様 *spec* を満足することを表す．振舞い仕様の入力（*spec.input.behavior*）と 振舞い（*target*）の並行合成が振舞い仕様の入出力（*spec.inputOutput.behavior*）を満足するとき，振舞い *target* が振舞い仕様 *spec* を満足するという．

3.3 階層モデル

本研究で対象とする階層モデルを定義する．階層モデルの概略を図 4 に示す．

コンポーネントには名前があり名前の集合を *NAME* とする．

$[NAME]$

コンポーネント（*Component*）は上位コンポーネント（*upper*），下位コンポーネント（*lower*），インターフェース（*interface*）およびコンポーネントの定義（*definition*）から構成される．

Component

$$upper : NAME; lower : \mathbb{P}\ NAME$$
$$interface : Specification; definition : Behavior$$

コンポーネントの上位コンポーネント（*upper*）は 1 つであり，下位コンポーネント（*lower*）は複数で名前の集合として表す．インターフェース（*interface*）は振舞い仕様，コンポーネントの定義（*definition*）は振舞いとして表す．

階層モデルを *HierarchicalModel* として定義する．

HierarchicalModel

$$hm : NAME \longrightarrow Component$$

$$\forall n : \mathrm{dom}(hm) \bullet (hm\ n).upper \in \mathrm{dom}(hm) \land (hm\ n).lower \subseteq \mathrm{dom}(hm)$$

hm は名前（*NAME*）からコンポーネント（*Component*）への関数であり，*hm* で定義されているコンポーネントの上位コンポーネントおよび下位コンポーネントは *hm* で定義されていることを表す．*hm* にはこれ以外に満たすべき性質があるがここでは紙面の都合上省略する．

3.4 コンポーネントの仕様

コンポーネントの仕様を *ComponentSpecification* として定義する．

ComponentSpecification

$$target : Component; spec : \mathbb{P}\ Specification$$

$$\forall s : spec \bullet target.interface.inputOutput.events = s.inputOutput.events \land (target.definition.behavior, s) \in satisfy$$
$$target.interface.inputOutput.behavior = choice(\{s : spec \bullet s.inputOutput.behavior\})$$

コンポーネント（*target*）の仕様（*spec*）は振舞い仕様（*Specification*）の集合として表す．*spec* のそれぞれの仕様（*s*）はインターフェースの入出力を表し，*target* の振舞い仕様として満足している．コンポーネントのインターフェースの振舞いは仕様（*spec*）の選択合成として表される．

3.5 網 羅 性

インターフェースの網羅性を *Exhaustiveness* として定義する．

Exhaustiveness

HierarchicalModel; *ComponentSpecification*

$$\exists\, lb : \mathbb{P}\, BEHAVIOR \mid lb = \{n : target.lower \bullet (hm\ n).interface.inputOutput.behavior\} \bullet (concurrent(\{target.interface.input.behavior, target.definition.behavior\} \cup lb), target.interface) \in satisfy$$

網羅性は，インターフェースの入力，コンポーネントの定義および下位コンポーネントのインターフェースの入出力の並行合成が，インターフェースの振舞いを満足することを表す．網羅性は，コンポーネントの仕様（*ComponentSpecification*）が，想定している上位コンポーネントからの入力と下位コンポーネント群の入出力の振舞いの合成から得られる振舞いであることを保証する．

4. CSP/FDR を用いた検証

振舞い検証のモデルに基づいて，プロセス代数 CSP のモデル検査器 FDR を用いた検証方法について述べる．システムを状態遷移機械の集合として表現するための CSP のライブラリを提示した後，検証の方法を示す．CSP の表記は FDR および 文献 2) で定義されている CSP のアスキーコードの表現 CSP_M を用いる．

4.1 CSP ライブラリ

まず，コンポーネント間のイベントの送受信のチャネル（snd,rcv）を定義する．

```
channel snd, rcv: StmName.EventName
SEND(to, event) = snd.to.event -> SKIP
RECV(stm, event) = rcv.stm.event -> SKIP
```

イベントの送受信は上記の SEND(),RECV() を用いて表現する．システムの表記を以下に示す．

```
SYSTEM(stms, env, sync_env, interface, sync_interface) =
  let
    QueueSync = {snd.stm.e, rcv.stm.e | stm <- stms, e <- EventName}
    StateSync = {get.stm.s, set.stm.s | stm <- stms, s <- StateName}
  within
    ((env [| sync_env |] (interface [| sync_interface |] STMs(stms)))
       [| QueueSync |] QUEUEs(stms))
     [| StateSync |]
    STATEs(stms)
```

システム（SYSTEM）は，外部環境（env），インターフェース（interface），状態遷移機械（STMs(stms)），イベントキュー（QUEUEs(stms)）および状態（STATEs(stms)）の並行合成として定義される．状態遷移機械，イベントキューおよび状態は，対象とする状態遷移機械に対して定義する．状態は状態遷移機械の現在の状態を保持する．状態遷移機械の定義を以下に示す．

```
STMs(stms) = ||| stm:stms @ STM(stm)
STM(stm) =
  get.stm?state:STM_STATE(stm) ->
    [] event: AcceptedEvents(stm, state) @
      (rcv.stm.event -> ACTION(stm, state, event);
        set.stm!StateTransition(stm, state, event) -> STM(stm))
```

状態遷移機械は，現在の状態を獲得してイベントキューからのイベントを待ち，イベントを受信した後でアクションを実行し状態を遷移させる．状態遷移機械は状態毎に，受信するイベント（AcceptedEvents），アクション（ACTION）および状態遷移（StateTransition）を定義する．イベントキューの定義を以下に示す．

```
QUEUEs(stms) = ||| stm:stms @ QUEUE(stm, <>)
QUEUE(stm, seq) =
  #seq < QUEUE_SIZE & ENQUEUE(stm, seq)
[]
  #seq > 0 & DEQUEUE(stm, seq)
ENQUEUE(stm, seq) = snd.stm?event -> QUEUE(stm, seq^<event>)
DEQUEUE(stm, seq) = rcv.stm!head(seq) -> QUEUE(stm, tail(seq))
```

イベントの送受信の意味は QUEUE で定義される．上記の QUEUE は FIFO キューを表している．状態の定義を以下に示す．

```
channel get, set: StmName.StateName
STATEs(stms) = ||| stm:stms @ STATE(stm, InitialState(stm))
STATE(stm, state) =
```

```
    get.stm!state -> STATE(stm, state)
  [] set.stm?to -> STATE(stm, to)
```

状態遷移機械の状態は get チャネルと set チャネルを用いて獲得および設定する．状態をプロセスとして表現することにより，状態遷移機械およびイベントキューで状態毎の記述を可能にしている．

4.2 検　　証

振舞い検証のモデルで定義した検証について CSP/FDR による検証方法を説明する．

4.2.1 振舞い仕様の検証

振舞い仕様は，検証の対象となる状態遷移機械への入力に対する入出力として与えられる．状態遷移機械への入力は SYSTEM の外部環境（env）として与える．検証対象の状態遷移機械の集合を Targets，外部環境と同期するイベントを SyncEnv，インタフェースの振舞いを INTERFACE，インタフェースと同期するイベントを SyncInterface として対象システム（TARGET）を以下のように定義する．

```
TARGET(env) = SYSTEM(Targets, env, SyncEnv, INTERFACE, SyncInterface)
```

外部環境およびインターフェースと同期するイベントの指定は，外部環境から状態遷移機械もしくはインターフェースの振舞いを制約する場合にそのイベントを指定する．

振舞い仕様の入力を INPUT，入力に対する入出力を Spec とすると，対象システムの振舞いが仕様を満たすことを CSP の詳細化関係[2] を用いて以下のように表現する．

```
assert SPEC [F= TARGET(INPUT) \ diff(EventsSystem, EventsSpec)
```

assert は モデル検査器 FDR のコマンドで詳細化関係の検査を実行する．[F= は CSP の失敗モデル（failure model）の詳細化関係を表す．失敗モデルは一般的に活性の性質を検証するのに用いられる．上記の詳細化関係は「TARGET に INPUT の入力を与えた振舞いが，SPEC のイベントに着目すると SPEC の振舞いとなる」ということを表している．EventsSystem はイベント全体，EventsSpec は Spec のイベントを表し，隠ぺい（\）を用いて TARGET のイベントを限定している．

4.2.2 コンポーネントの仕様

コンポーネントの仕様は振舞い仕様の集合として表され，振舞い仕様の選択合成が対象コンポーネントのインターフェースを表す．振舞い仕様の入力を INPUT(i)，入力に対する入出力を SPEC(i) として表し，仕様の集合を Spec とすればインタフェースは以下のように表される．

```
INTERFACE_INPUT = [] i:Spec @ INPUT(i)
INTERFACE_SPEC = |~| i:Spec @ SPEC(i)
```

上記の [] は外部選択（external choice）の合成，|~| は内部選択（internal choice）の合成を表す．コンポーネントへの入力に対する入出力の振舞いは非決定的に起こるので，入出力の仕様は内部選択の合成となる．

4.2.3 網羅性の検証

網羅性は，インターフェースの入力，コンポーネントの定義および下位コンポーネントのインターフェースの入出力の並行合成が，インターフェースの振舞いを満足することを表す．網羅性の検証は以下のシステムに対して行う．

```
TARGET_INTERFACE(env) = SYSTEM(Targets, env, {}, INTERFACE, Sync)
```

TARGET_INTERFACE では，外部環境と同期するイベントの指定は空集合（{}）となる．網羅性の検証は以下のように表される．

```
assert INTEFACE_SPEC [F= TARGET_INTEFACE(INTERFACE_INPUT)
                                    \ diff(EventsSystem, EventsInterface)
```

外部環境と同期するイベントの指定が空集合であることが，TARGET_INTERFACE の振舞いが想定している状態遷移機械とインターフェースの全ての組み合わせであることを示している．

5. 事　　例

複合スイッチと自動販売機を事例として振舞い検証の具体例を示し，検証の意義について考察する．

5.1 複合スイッチ

図 1 で構成された複合スイッチにおいて，スイッチ 1（SW1）と 複合スイッチ（CompSW）のインターフェースの検証について説明する．

5.1.1 スイッチ 1（SW1）

第 2.2 節で示した SW1 のインターフェースについて考える．インターフェースの入力は以下のようになる．

```
INTERFACE_SW1_ENV =
  SEND(SW1, enable) [] SEND(SW1, enable); SEND(SW1, disable)
```

CompSW から SW1 へは enbale の後に disable が送信される場合がある．上記の入力 INTERFACE_SW1_ENV に対する入出力は以下のようになる．

```
INTERFACE_SW1 =
```

```
SEND(SW1, enable); SEND(CompSW, sw1Pressed) |~|
SEND(SW1, enable); SEND(SW1, disable); SEND(CompSW, ack) |~|
SEND(SW1, enable); SEND(SW1, disable); SEND(CompSW, sw1Pressed);
                                        SEND(CompSW, ack) |~|
SEND(SW1, enable); SEND(CompSW, sw1Pressed); SEND(SW1, disable);
                                        SEND(CompSW, ack)
```

また，ハードウェアからの通知はインターフェースとして SW1_DEVICE として表される．

```
SWITCH1_PRESSED = SEND(Switch1, pressed)
SW1_DEVICE = SWITCH1_PRESSED; INTERFACE_SW1_DEVICE
```

SW1 のインターフェースを検証するためのシステムを以下に示す．

```
TARGET_INTERFACE(env) = SYSTEM({SW1}, env, {}, SW1_DEVICE, {})
```

SW1 のインターフェースの網羅性の検証は以下のようになる．

```
assert INTEFACE_SW1 [F= TARGET_INTEFACE(INTERFACE_SW1_ENV)
                                \ diff(EventsSystem, EventsInterface)
```

SW1 への INTERFACE_SW1_ENV に対する CompSW への振舞いは INTEFACE_SW1 であることを検証している．

5.1.2 複合スイッチ（CompSW）

CompSW のインターフェースの入力は以下のようになる．

```
INTERFACE_CompSW_ENV = SEND(CompSW, enable)
```

上記の入力 INTERFACE_CompSW_ENV に対する入出力は以下のようになる．

```
INTERFACE_CompSW =
  SEND(CompSW, enable);
  (SEND(Caller, sw1Pressed) |~| SEND(Caller, sw2Pressed))
```

下位コンポーネント（SW1 と SW2）の振舞いは，下位コンポーネントのインターフェースを用いる．

```
LOWER_COMP = INTERFACE_SW1 ||| INTERFACE_SW2
```

CompSW のインターフェースを検証するためのシステムを以下に示す．

```
TARGET_INTERFACE(env) = SYSTEM({CompSW}, env, {}, LOWER_COMP, Sync)
```

Sync は SW1 および SW2 への送信イベント（enable, disable）である．これらのイベントを同期することにより，CompSW と SW1 および SW2 の間の振舞いを規定する．CompSW のインターフェースの網羅性の検証は以下のようになる．

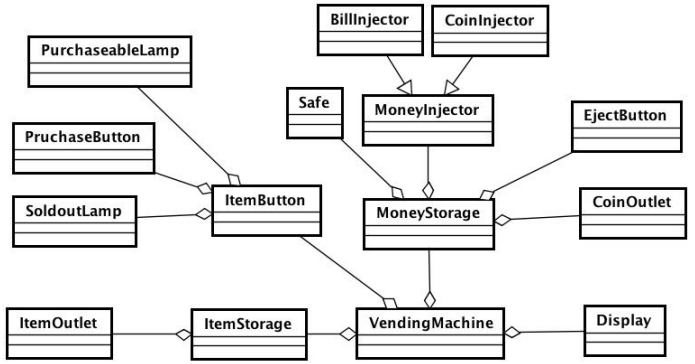

図 5 自動販売機のクラス図
Fig. 5 The class diagram for a vending machine

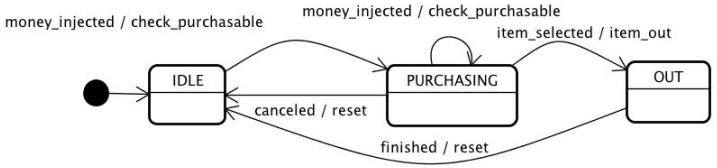

図 6 自動販売機の状態遷移図
Fig. 6 The state transition diagram for the vending machine

```
assert INTEFACE_CompSW [F= TARGET_INTEFACE(INTERFACE_CompSW_ENV)
                                \ diff(EventsSystem, EventsInterface)
```

CompSW への INTERFACE_CompSW_ENV に対する上位コンポーネントへの振舞いは INTEFACE_CompSW であることを検証している．

5.2 自動販売機

自動販売機を事例としてインターフェースの仕様を考える．自動販売機のクラス図を図 5 に示す．自動販売機（VendingMachin）は商品ボタン（ItemButton），硬貨入れ（CoinInjector）および商品排出（ItemOutlet）などから構成される．

自動販売機の販売の状態遷移図を図 6 に示す．自動販売機はアイドル（IDLE）の状態からお金が投入される（money_injected）と購入可能な商品を調べて（check_purchasable）状態

を購入中（PURCHASING）に遷移する．購入中の状態で商品が選択される（item_selected）と商品の排出を指示（item_out）して状態を商品排出中（OUT）に遷移する．商品の排出が完了する（finished）と状態をアイドル（IDLE）に遷移する．

例えば，状態が PURCHASING のとき商品の選択（item_selected）と返却レバーからの購入キャンセル（canceled）のイベントを待つ．これをボタンと返却レバーの複合機械が制御すると考える．この複合機械は前述の複合ボタンと同様の構造となり，インターフェースは以下のようになる．

```
INTERFACE_ButtonLever_ENV = SEND(ButtonLever, enable)
```

上記の入力 INTERFACE_CompSW_ENV に対する入出力は以下のようになる．

```
INTERFACE_ButtonLever =
  SEND(ButtonLever, enable);
  (SEND(VendingMachine, item\_selected) |~|
  SEND(VendingMachine, canceled))
```

自動販売機の状態ごとにイベントを待つ複合機械を持つ階層構造を構成すれば，複合スイッチと同じ構造で検証をすることができる．

5.3 検証の意義

振舞い検証のモデルに基づいた網羅性の検証の意義として，

- コンポーネント記述の信頼性向上
- コンポーネント間のプロトコルを陽に記述
- コンポーネント間の不整合がないことの保証

があげられる．

コンポーネントのインターフェースを記述して検証することは，モデル検査器を用いてコンポーネントをテストもしくは実行することに相当する．インターフェースの仕様は選択合成で表現されており，これはテストケースを作成することに相当する．網羅性の検証は，テストケースが十分であることの形式的な基準を提示している．

5.4 ツールによる支援

本稿では CSP の記述は全て手作業で行った．手作業では入力の誤りなど多くのコストがかかる．検証のコストを削減するために以下のツールの開発が考えられる．

- 状態遷移機械の CSP 記述の自動生成
- 振舞い仕様の図式表現とエディタ
- 検証条件の自動生成

上記のツールの開発とともに実用規模のシステム開発における本検証の適用を試みる予定である．

6. おわりに

本稿では，階層的に分割されたシステムのアーキテクチャ記述に対して，網羅性に着目した検証モデルを提案し，プロセス代数 CSP のモデル検査器 FDR を用いて検証する方法を提示した．網羅性の検証は，コンポーネント間のインターフェースの記述が，下位コンポーネントで想定するイベントの全ての組み合わせに対する振舞いを表現にしていることを保証する．このことが，下位コンポーネント群の振舞いをインターフェースで置き換えることを可能にし，モデル検査における状態爆発を回避する手段となる．また，検証手法の提示は検証の指針となり得る．

我々は組込みシステムのためのアスペクト指向ソフトウェアアーキテクチャスタイル E-AoSAS++[6),7)] の研究を行っている．E-AoSAS++では横断的な関心事をモジュール化することによりモジュールの再利用性が向上する反面，それらの合成が正しく振舞う事を検証する必要がある．今後，E-AoSAS++ のアーキテクチャ記述に対して本手法を適用する予定である．

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），および平成 22 年度南山大学パッヘ奨励金 I-A-1，1-A-2 の助成を受けて実施した．

参 考 文 献

1) 平山雅之: 組み込みソフトウェア開発の現状, 情報処理, Vol. 45, No. 7, pp. 667-681, 2004.
2) A. Roscoe: The Theory and Practice of Concurrency, Prentice Hall, 1998.
3) Formal Systems Limited, *http://www.fsel.com/*.
4) M. Shaw and D. Garlan: Software Architecture, Prentice Hall, 1996.
5) M. Spivaey: the Z Notation: A Reference Manual, 2nd edition, Prentice Hall, 1992.
6) M. Noro, A. Sawada, Y. Hachisu, and M. Banno: E-AOSAS++ and its Software Development Environment, Proceedings of the 14th Asia-Pacific Software Engineering Conference(APSEC2007), pp. 206-213, 2007.
7) 加藤大地, 蜂巣吉成, 沢田篤史, 野呂昌満: E-AOSAS++ に基づく開発支援環境 – 実行前検査ツールの提案 –, 情報処理学会技術研究報告, ソフトウェア工学研究会 (2009-SE-163), Vol. 2009, No. 31, pp. 121-128, 2009.