Vol.2014-SLDM-165 No.8
Vol.2014-EMB-32 No.8
2014/3/15

際どい実行順序を考慮した並行システム検証に関する考察

張 漢明[1,a)] 沢田 篤史[1,b)] 野呂 昌満[1,c)]

概要：本研究の目的は，並行システムにおける際どい実行順序を考慮した検証手法を提示することである．本研究の基本的なアイデアは，際どい実行順序とその仕様をプロセス代数 CSP を用いて定式化して，系統的な検証式の生成手順を提示することである．本稿では，際どい実行順序として，複数のセンサーを用いた事象の検知を取り上げ，同時に事象を検知することを考慮した検証方法を示し，自動販売機システムへの事例適用とその有用性について議論する．際どい実行順序の定式化は，その検証作業の効率化と，設計の後工程における仕様として有用である．

A discussion on the verification of concurrent systems for considering critical execution order

Han-Myung CHANG[1,a)] Atsushi SAWADA[1,b)] Masami NORO[1,c)]

Abstract: We aim to establish verification methods of concurrent systems for considering critical execution order. Our basic idea is to formalize the critical execution order and its specification using the process algebra CSP, and to show how to make an assertion expression systematically in order to verify it. In this paper we pay attention to detecting simultaneous events with sensors as the critical execution order, and give the verification methods using a simple vending machine example. Formalizing the critical execution order is effective as a specification after the design phase.

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．システム開発の設計工程における並行システムの検証において，モデル検査の有用性が報告されている [1], [7]．モデル検査では，並行システムの状態を網羅的に検査することにより，予期せぬ振る舞いや状態を検出することができる．実際のソフトウェア開発にモデル検査を普及するための課題として，系統的な検証手法の提示があげられる．

ステートチャートや UML 等の図式表現を，モデル検査器を用いて検証する試みが行われている．Roscoe らはステートチャートをプロセス代数 CSP[9] に変換して，モデル検査器 FDR[3] を用いて競合状態（race conditions）などのエラーを検出している [8]．Hsiung らは組込みシステム向けの UML モデルからモデル検査を行うためのフレームワークを提供している [4]．これらは，汎用の検証の枠組みを与えているが，検証の基準となる仕様をどのようにして得るかが課題となる．

本研究の目的は，並行システムにおける際どい実行順序を考慮した検証手法を提示することである．逐次実行プロセスでは同じ入力に対する実行順序は同じであるが，並行システムでは同じ入力に対する実行順序は一般的に非決定的であり，その振る舞いを完全に記述することは本質的に困難な作業である．特に，非同期通信の計算モデルでは，その振る舞いは複雑であり，予期せぬ際どい実行順序の存在がないことをレビューで示すことは難しい．際どい実行順序の本質を分析して，その概念と検証方法を提示することが本研究の目標である．

本研究の基本的なアイデアは，際どい実行順序とその仕様をプロセス代数 CSP を用いて定式化して，検証式の生成手順を提示することである．並行システムを並行に動作する状態遷移機械の集合と捉え，非同期通信による状態遷

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) sawada@nanzan-u.ac.jp
c) yoshie@nanzan-u.ac.jp

© 2014 Information Processing Society of Japan

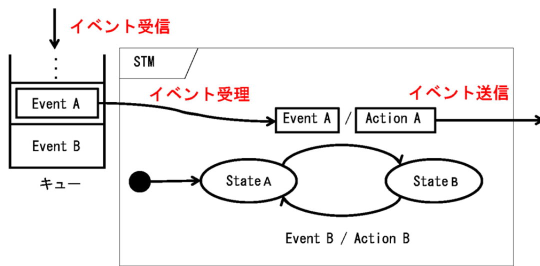

図 1 並行状態遷移機械

Fig. 1 Concurrent state transition machine

移機械間の相互作用に着目する．CSP はイベントを用いてプロセス間の相互作用の記述に適しており，モデル検査器 FDR を用いて仕様と実現の間の詳細化関係（Refinement）を自動で検証することができる．対象システムとその入出力を表す環境との並行合成が，仕様を満たすことを検証式で表す．所望の振る舞いを CSP を用いて定式化するには，対象とする振る舞いの本質的なイベントの同定と，並行合成を用いたプロセスのモジュール化が鍵となる．

本稿では，際どい実行順序として，複数のセンサーを用いた事象の検知を取り上げ，同時に事象を検知することを考慮した検証方法を示し，自動販売機システムへの事例適用とその有用性について議論する．並行システムでは「同時」について考慮することは避けられない．通常，1 つの事象が起こった場合の処理は問題ない．複数の事象が同時に起こることが際どいタイミングで稀な場合なので，これをテストで検証することは困難である．

ここでは，複数の事象を同時に検知することを「同時事象検知」として，その検証方法を提示する．事象を検知するセンサーのコントローラが複数あることを想定して，同時事象検知の実行順序を，関連するイベントをパラメータとして CSP 式で定式化した．対象システムにこの同時事象検知を制約として並行合成することにより，同時に事象を検知した全ての振る舞いを記述することができる．検証事例として，自動販売機システムにおいて，購入ボタンと返却レバーを同時に押した際に，購入ボタンを優先して商品を購入することを検証した．

2. 基本的なアイデア

本章では，本研究で対象とする並行計算モデルと検証の枠組みについて説明し，本稿で取り上げる際どい実行順序として，同時事象検知について説明する．

2.1 対象とする並行計算モデル

本研究では，並行システムを

- 並行に動作する状態遷移機械の集合

としたコンポーネントを基本とするモデル化を前提とする．コンポーネント間の通信は，組み込みシステムで一般的な非同期通信モデルとする．ここでは，システムの設計段階の記述を想定して，コンポーネント間のイベントの相互作用に着目する．

並行状態遷移機械の概念を図 1 に示す．状態遷移機械の遷移にはイベントとアクションの組が書かれている．キューからのイベントを受理してアクションを実行する．イベント受理はキューの先頭から一致するイベントを走査する．アクションでは，コンポーネント間の相互作用に着目するので，状態遷移機械へのイベント送信を記述する．

```
channel send, recv: Stm.Event

CH_FIFO(stm, seq, size, state) =
  #seq < size & send.stm?event ->
            CH_FIFO(stm,seq^<event>,size,state)
[]
  #seq > 0 & RECV_FIFO(stm,seq,size,state,<>)

RECV_FIFO(stm,<>, size, state, _) = STOP
RECV_FIFO(stm,<event>^seq,size,state,front) =
  if member(event,AcceptedEvents(stm, state))
  then recv.stm!event ->
          CH_FIFO(stm, front^seq, size,
             StateTransition(stm,state,event))
  else RECV_FIFO(stm, seq, size,
             state, front^<event>)
```

図 2 CSP によるキューの定義

Fig. 2 Definitions of the queue in CSP

2.2 検証の枠組み

本研究では，状態遷移機械をプロセス代数 CSP の記述に変換して，CSP の代表的なモデル検査器 FDR を用いて検証する [2]．CSP はイベントに基づいた並行システムの形式言語であり，仕様と実現の関係を詳細化関係（Refinement）を用いて記述する．FDR はこの詳細化関係を自動で検証することができる．

対象となるシステムは状態遷移機械，環境，，キューから構成される．環境はシステムの入出力を表し，組み込みシステムでは，環境はデバイスドライバに相当する．CSP は同期通信モデルなので，非同期通信のキューを定義する必要がある．CSP によるキューの定義を図 2 に示す．

これらの構成を CSP を用いて SYSTEM_() として定義する．

```
SYSTEM_(env, sync, stms, chs, events) =
      (env [| sync |] stms) [| events |] chs
```

SYSTEM_() の引数のうち，env, stms, chs はそれぞれ環境，状態遷移機械，キューのプロセスである．sync は環境と状態遷移機械の間の同期イベント，events は状態遷移機械とキューの間の同期イベントである．状態遷移機械とキューはインターリーブ合成を用いて，以下のように定義する．

```
||| stm:TargetSTM @ STM(stm, Init(stm))
||| stm:TargetSTM @
```

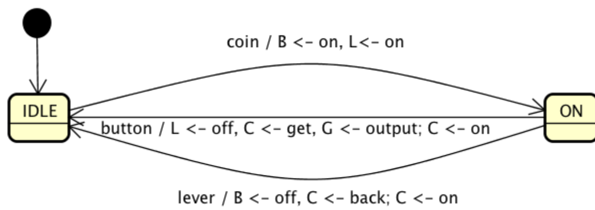

図 3　自動販売機の制御
Fig. 3　Vending machine controller

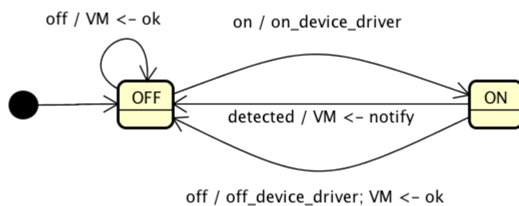

図 4　センサー
Fig. 4　Sensor

CH_FIFO(stm, <>, BUF_SIZE, Init(stm))

TargetSTM は状態遷移機械の集合で，状態遷移機械 STM とキュー CH_FIFO を並行に動作させている．STM は以下の SEND() と RECV() を用いて定義する．

```
SEND(stm, event) = send.stm.event -> SKIP
RECV(stm, event) = recv.stm.event -> SKIP
```

SEND() は stm へのイベント送信，RECV() は stm へのイベント受理を表す．

2.3　同時事象検知

際どい実行順序として同時事象検知について説明する．並行システムでは同時に事象が起こることを避けることができない．例えば，複数のセンサーがあるシステムで，あるハードウェアのセンサーの事象を検知した場合，その処理をする前に，他のハードウェアのセンサーが事象を通知する可能性がある．ここでは，このような場合を同時事象検知という．

単純な自動販売機システムを例として同時事象検知を示す．図 3 は自動販売機制御の状態遷移機械である．IDLE の状態からコイン（coin）が投入されると，ボタンと返却レバーに on イベントを送信（B <- on, L <- on）して，ボタンと返却レバーを作動させる．ボタンが押される（button）と商品を排出（G <- output）し，返却レバーが引かれると（lever）とコインを返却（C <- back）する．

ボタンと返却レバーの状態遷移機械を図 4 に示す．OFF の状態で on イベントを受信すると，ハードウェアの事象検知を可能とする．ハードウェアが事象を検知すると detected イベントを受信して，ハードウェアの事象検知を停止し，自動販売機制御に notify イベントを送信する．ボタンの場合，detected イベントは pushed イベント，

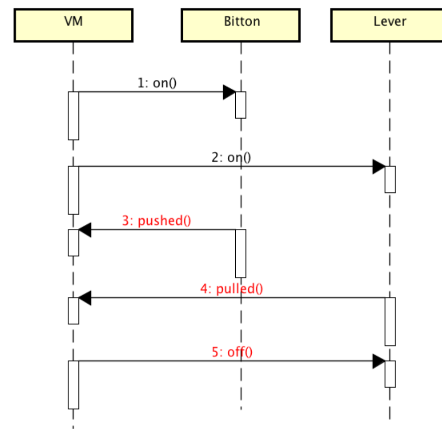

図 5　同時事象検出の例
Fig. 5　An example of detecting simultaneous events

notify イベントは button イベントとなり，返却レバーの場合，detected イベントは pulled イベント，notify イベントは lever イベントとなる．

同時事象検知の実行例を図 5 に示す．ボタンと返却レバーに on イベントが送信された後，ボタンから pushed イベントが送信されている．自動販売機制御は返却レバーに off イベントを送信するが，off イベントが送信される前に，返却レバーから pulled イベントが送信されている．このような状態になることが同時事象検知である．非同期通信モデルの場合，off イベントを送信した後で，pulled イベントが送信される場合も同時事象検知である．センサーの数が多くなると，同時事象検知を網羅することは容易ではないので，これを CSP 式で定式化する．

3.　同時事象検知の定式化

本章では，事象を同時に検知する「同時事象検知」に関連するイベントを示して，同時事象検知の振る舞いに関連するイベントをパラメータとして CSP 式を定義する．

3.1　同時事象検知に関連するイベント

事象の検知には，事象を検知する複数の「センサー」とセンサーを制御する「コントローラ」がある．センサーはデバイスドライバを用いて物理的なハードウェアを制御する．センサーの状態遷移機械は前節で示した図 4 である．通常の事象の検知のイベントの実行順序は以下の通りである．

(1) ハードウェアが事象を検知
(2) デバイスドライバがセンサーに detected イベントを送信
(3) センサーがコントローラに notify イベントを送信
(4) 検知したセンサー以外のセンサーにコントローラが off イベントを送信

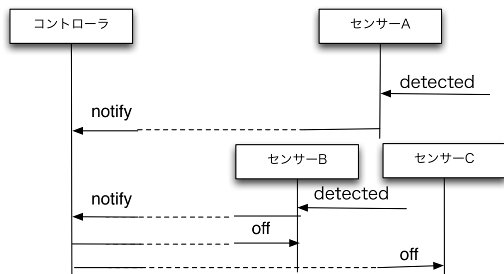

図 6 同時事象検知に関連するイベント
Fig. 6 Events related to detecting simultaneous events

コントローラは事象を検知すると，一旦全てのセンサーを停止してからその事象の処理をする，というのが制御の基本的な方針である，コントローラが，あるセンサーの事象を検知して，他のセンサーを停止する前に，他のセンサーが事象を検知している状態が「同時事象検知」となる．

同時事象検知に関連するイベントを図 6 に示す．事象検知に関わるイベントは detected，notify，off の 3 種類のイベントである．

- detected は，デバイスドライバからセンサーへの送信イベントである．
- notify は，センサーとコントローラの間に他のコンポーネントがあるかもしれないこと（図の破線）を考慮すれば，コントローラの受信イベントが適切である．
- off は，notify と同様に，センサーの受信イベントが適切である．

したがって，以下の 3 種類のイベントが同時事象検知に関連するイベントである．

detected　デバイスドライバの送信イベント
notify　コントローラの受信イベント
off　センサーの受信イベント

3.2 同時事象検知の定義

同時事象の検知は，コントローラが notify イベントを受信して，他のイベントに off イベントを送信する間に，他のコントローラが notify イベントを送信している場合に起こる．したがって，最初の事象の検知は，ハードウェアの検知ではなく，コントローラの notify イベント受信で認識される．

本節では，同時事象検知に関わるイベントをセンサーイベントとして定義して，まず，コントローラが notify イベントを最初に受信した場合の同時事象検知 CONFLICT_F を定義する．その後，同時事象のセンサーを指定した CONFLICT_S，指定したセンサーを含む全ての同時事象検知 CONFLICT_I を定義する．

3.2.1 センサーイベント

前節で述べた同時事象検知に関わる 3 種類のイベントをセンサーイベントとして以下の 3 項組で定義する．

```
(detected, notify, off)
```

例えば，ボタンのセンサーイベントは

```
Button = (send.button.pushed,
          recv.vm.button,
          recv.button.off)
```

となる．send.button.pushed はボタンへの pushed イベント送信，recv.vm.button は自動販売機コントローラでの button イベント受信，recv.button.off はボタンでの off イベント受信を表す．

3.2.2 CONFLICT_F

コントローラが notify イベントを最初に受信した場合の同時事象検知 CONFLICT_F を定義する．

```
CONFLICT_F(sensors,first,conflicts) =
    DETECTED({first}); NOTIFY({first});
      NOTIFY(diff(conflicts,{first}))
  |||
    DETECTED_OFF(diff(conflicts,{first}))
  |||
    OFF(diff(sensors,conflicts))
```

CONFLICT_F の引数は

sensors: センサーイベントの集合
first: 最初に検知するセンサーイベント
conflicts:（first を含む）同時に事象を検知したセンサーイベントの集合

である．定義の中で; はプロセスの逐次実行，||| はプロセスのインタリーブ合成を表している．diff(s1,s2) は集合 s1 から s2 の差集合である．

DETECTED(ss)，NOTIFY(ss)，OFF(ss) は，センサーイベントの集合 ss の，センサーへの検知イベント送信 (detected), コントローラでの通知イベント受信 (notify), センサーでの検知停止イベント受信（off），が，それぞれ順不同で起こることを表している．具体的な定義を以下に示す．

```
DETECTED(ss) = |||(d,_,_):ss @ d -> SKIP
NOTIFY(ss) = |||(_,e,_):ss @ e -> SKIP
OFF(ss) = |||(_,_,o):ss @ o -> SKIP
```

CONFLICT_F は 3 つのプロセスのインタリーブ合成で表されている．最初のプロセスは notify イベントの順序関係を定義している．最初に受信した first の notify イベントの後は，同時を検知した notify イベントが順不同で起こることを表している．また，first の notify イベントの前に detected イベントが起こることが必要である．

2 番目のプロセスは DETECTED_OFF(ss) を用いて first のセンサー以外の detected イベントと off イベントの順序関係を定義している．各センサでは detected イベントの後に off イベントが起こることを保持して，センサー間では順不同で起こることを表している．DETECTED_OFF(ss) の具体的な定義を以下に示す．

ⓒ 2014 Information Processing Society of Japan

```
DETECTED_OFF(ss) =
         |||(d,_,o):ss @ d -> o -> SKIP
```

最後に3番目のプロセスは検知されていないセンサーのイベントを定義している．センサーが検知していない場合は，off イベントだけが起こるので，事象を検知した以外のセンサーで off イベントが順不同で起こることを表している．

3.2.3 CONFLICT_S

同時に事象を検知したセンサーを指定する同時事象検知 CONFLICT_S を定義する．

```
CONFLICT_S(sensors,conflicts) =
  []first:conflicts @
    CONFLICT_F(sensors,first,conflicts)
```

CONFLICT_S の引数は

sensors: センサーイベントの集合

conflicts: 同時に事象を検知したセンサーイベントの集合

である．CONFLICT_S は同時に事象を検知した conflicts のうち，いずれかのセンサーが最初に認識されたものであることを CONFLICT_F を用いて定義している．

3.2.4 CONFLICT_I

指定したセンサーを含む全ての同時事象検知 CONFLICT_I を定義する．

```
CONFLICT_I(sensors,conf) =
  let
    ss = {s | s<-Set(sensors), conf<-s}
  within
    []s:ss @ CONFLICT_S(sensors,s)
```

CONFLICT_I の引数は

sensors: センサーイベントの集合

conf: 同時事象に指定するセンサーイベント

である．Set(s) は s の部分集合の集合であり，CONFLICT_I は指定された同時事象 conf を含んだセンサー（sensors）の集合の集合（ss）を作成して，CONFLICT_S を用いて定義している．

4. 同時事象検知の検証

同時事象検知の検証例を自動販売機システムを用いて示す．ここでは，ボタンが押されると必ず商品を購入するという仕様を想定する．図3の状態遷移機械は同時事象検知に対応していない．ボタン，返却レバーと自動販売機制御の間に，図7に示した同時事象を検知する状態遷移機械を挟むことにする．この状態遷移機械を conf とすると，ボタンと返却レバーのセンサーイベントは以下のようになる．

```
Button = (send.button.pushed,
          recv.conf.button,
          recv.button.off)
Lever =  (send.lever.pulled,
          recv.conf.lever,
          recv.lever.off)
```

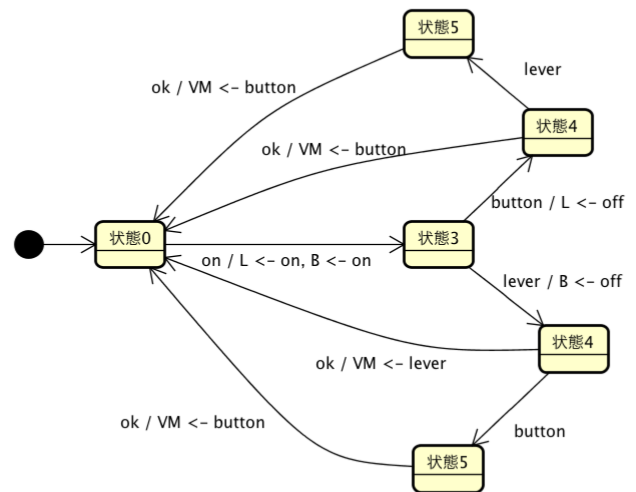

図 7 同時事象検知する状態遷移機械
Fig. 7 STM for detecting simultaneous events

ボタン押下を含んだ同時事象は CONLICT_I を用いて以下のように定義できる．

```
CONF_BUTTON =
     CONFLICT_I({Button,Lever},Button)
```

自動販売機の振る舞いを SYSTEM として，CONF_BUTTON の制約を，センサーイベントを同期イベントとする並行合成を用いて，以下のように定義することができる．

```
SYSTEM_B = SYSTEM [|SY1|] CONF_BUTTON
SY1 = union(Button,Lever)
```

仕様は，着目するイベントに対して，その振る舞いを定義することで記述する．ここでは，自動販売機制御への button イベントと lever イベントに着目して，button イベントが送信されることを確認すれば良い．従って，仕様 SPEC を以下のように定義する．

```
SPEC = send.vm.pushed -> SPEC
```

仕様と対象システムの関係は，詳細化関係 [F=を用いて以下のようになる．

```
SPEC [F= SYSTEM_B \ diff(Events,SY2)
SY2 = {send.vm.button,send.vm.lever}
```

対象システム SYSTEM_B は，隠蔽\を用いて限定している．Events は全てのイベントを表している．

このシステムにボタンが押される時間をタイマーで監視した場合，タイマーの状態遷移機械も図4で示したセンサーとみることができる．タイマーのセンサーイベントを

```
Timer =  (send.timer.timeout,
          recv.conf.timer,
          recv.timer.off)
```

とすると，このセンサーイベントを追加して，同時イベント検知を定義することができる．

```
CONF_WITH_TIMER =
```

ⓒ 2014 Information Processing Society of Japan

```
CONFLICT_I({Button,Lever,Timer},Button)
```

このようにセンサーを指定する事により，同時事象検知の振る舞いを記述することができる．

5. 考察

本章では，本手法の一般性と仕様としての設計文書について考察する．

5.1 本手法の一般性

同時事象検知は，ある状態で複数のイベントを待つ場合に起こる可能性がある．本手法では，複数のセンサーの検知を，図4の状態遷移機械を用いて制御する方法について述べた．前章で示した通り，タイマーの監視も図4の状態遷移機械を用いることができる．このセンサーを用いる限り前章で定義した同時事象検知の関数を用いて，その振る舞いを記述することができる．

一般的には，本稿で示した同時事象検知をそのまま用いることができない．しかし，同時の事象を検知するためのセンサーイベントのうち，ハードウェアの検出（detected）とコントローラへの通知（notify）に相当するイベントは，どのようなシステムでも同時を検出するためには必ず必要である．これらの順序関係は

```
CONFLICT(sensors,first,conflicts) =
    DETECTED({first}); NOTIFY({first});
      NOTIFY(diff(conflicts,{first}))
  |||
    DETECTED(diff(conflicts,{first}))
```

として表すことができる．この関係以外の制約を加えることにより，ドメインに依存した同時事象検知を定義することができる．

5.2 仕様としての設計文書

本稿で示した検証の枠組みでは，対象システムの「状態遷移機械」と「環境」を記述する必要がある．状態遷移機械の記述では，アクションにおいて送信イベントを定義した．設計段階では，送信イベントを非決定的に記述することにより，抽象度の高い記述が可能である．この記述を仕様として，イベント送信に関するプログラムを半自動的に生成することにより，設計時の検証が有用なものになると考えられる．

また，環境の記述は，組み込みシステムの場合は，デバイスドライバに相当する．例えば本稿で用いた自動販売機のボタンの環境は

```
DD = (recv.button.on ->
          (send.button.pushed -> DD []
           send.button.off -> DD))[]
       send.button.off -> DD
```

と定義することができる．これは設計時の検証を有効にするための，デバイスドライバの仕様として重要な記述となる．

6. おわりに

本稿では，並行システムにおける際どい実行順序として，同時に事象を検出した場合の振る舞いの定式化とその検証方法を事例を用いて示した．並行システム，特に非同期通信のモデルでは，「同時」に事象が起こる事を考慮することは避けられない．ここでの「同時」はエラーではなく，同時に事象が起こった場合の仕様があり，これを検証することが重要である．

複数のセンサーがあるシステムに対して，同時に事象を検知した場合の振る舞いをイベントをパラメータとしたCSP式で定義した．CSP式の定義は，同時事象検知の意味を明確にするとともに，同時事象が起こるシステムの振る舞いを，システムの制約として記述することを可能にした．今後の研究課題として，実用規模のシステムの設計に対して本手法を適用することがあげられる．

謝辞 本研究の一部は，JSPS科研費 24500049，24220001，2013年度南山大学パッヘ奨励金 I-A-2 の助成を受けて実施した．

参考文献

[1] M. Ben-Ari: Principles of Concurrent and Distributed Programming 2nd edition, Addison-Wesley (2006).
[2] 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, pp. 107–112, 近代科学社 (2010).
[3] FDR2 Manual, Formal Systems Limited.
[4] P.A. Hsiung, C.W. Lin, C.H. Tseng, T.V. Lee, 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).
[5] K. Madhukar, R. Metta, P. Singh, R. Venkatesh: Reachability Verification of Rhapsody Statecharts, Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference, pp. 96–101 (2013).
[6] G. Ning: A Conponent-Based Software Development Model, COMPSAC'96, pp. 389–394 (1996)
[7] 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, pp. 72–86 (2006).
[8] A.W. Roscoe and Z. Wu: Verifying Statemate Statecharts Using CSP and FDR, Formal Methods and Software Engineering, Lecture Notes in Computer Science, 4260, pp. 324–341, Springer (2006).
[9] A.W. Roscoe: Understanding Concurrent Systems, Springer (2010).
[10] Software Architecture, http://www.sei.cmu.edu/architecture/.