UML ステートダイアグラムの亜種を用いた組み込みソフトウェア振舞い解析

中島 震[†,††]

組み込みソフトウェアの振舞い仕様のデザイン表現として UML ステートダイアグラムを用いることを考える．UML 標準の動作規則は組み込みソフトウェアが持つイベントの多様な属性，スケジューリング，といった特性を考慮していない．UML ステートダイアグラムの標準動作規則を変更して，組み込みソフトウェアが持つ特性を容易に表現する新たな規則を提案する．さらに，提案規則に従って動作するモデル記述を Promela に変換することで，モデル検査ツール SPIN を用いた性質検証が可能であることを示す．

Behavioral Analysis of Embedded Software Using a Variant of UML State-diagram

Shin Nakajima[†,††]

State-Transition Systems are employed to represent behavioural specification of embedded software. State Diagram of UML, having an execution semantics, is not adequate in regard to the semantics of the event system and the state-machine scheduling. This paper proposes a small extension to the semantics of the core part of UML State Diagram to perform design verification using the model-checking techniques.

1. はじめに

組み込みシステム開発では，ソフトウェアデザインの品質向上の技術として，形式仕様を適用する方法に期待が集まっている[7),8)]．特に，日常のソフトウェア開発の場ではモデリング言語 UML が標準的に使われるようになってきており，UML で表現したデザイン記述に，モデル検査に基づく検証方法を適用する研究が注目されている．

組み込みソフトウェアは多様であるが実世界と相互作用を持つことが共通点であり，実世界が持つ特徴を引き継ぐ[10)]．並行性・リアクティブ性・活性（liveness）という 3 つの特徴を持つ．そこで，状態遷移システムの考え方でデザインを表現する方法が有力であり[16)]，UML ではステートダイアグラムが状態遷移の考え方を提供する記法である．ところが，UML の標準動作規則は，組み込みソフトウェアが持つイベントの多様な属性やスケジューリングといった特性を考慮していない．

本稿では，UML ステートダイアグラムの標準動作規則を変更して，組み込みソフトウェアが持つ特性を容易に表現する新たな規則を提案する．さらに，提案規則に従って動作するモデル記述を Promela に変換することで，モデル検査ツール SPIN[6)] を用いた性質検証が可能であることを示す．

2. 状態遷移システムを用いるデザイン

組み込みシステムの特徴を概念的に整理する．次に，UML ステートダイアグラムの標準動作規則を示し，イベントと状態進行に関わるプライオリティ概念が重要であることを指摘する．

2.1 組み込みソフトウェアの特徴

組み込みソフトウェアは個別性が高く，その特徴を一言で述べることが難しい．実世界と相互作用を持つことが共通点であり，その結果，実世界が持つ特徴を引き継ぐ[10)]．

組み込みソフトウェアのデザイン表現を計算の仕組みの観点から整理すると，並行性・リアクティブ性・活性（liveness）という 3 つの特徴が浮かび上がる．第

† 情報・システム研究機構国立情報学研究所
National Institute of Informatics, Research Organization of Information and Systems
†† 科学技術振興機構
Japan Science and Technology Corporation

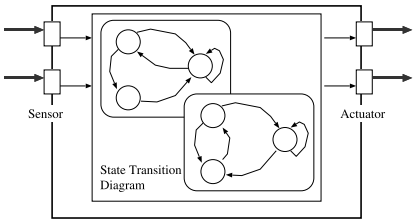

図 1 組み込みシステムの概念構成
Fig. 1 Embedded system model.

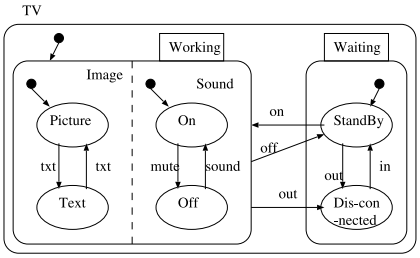

図 2 Statechart の記述例
Fig. 2 Statechart example.

1 に，実世界では，通常，複数の事象が同時に発生する．複数事象を処理するためには並行性を持たなければならない．第 2 に，事象に対応して何らかの処理を起こすリアクティブ性を持つ必要がある．同じ事象であっても，システムの状態に依存して処理が異なる．第 3 に，発生した個々の事象に対する計算結果の正しさだけでは不十分である．全体処理がつねに進行し発生事象に適切に呼応した処理を行う活性を満たさなければならない．

本稿で考える組み込みシステムの概念構成を**図 1** に示す．組み込みソフトウェアは装置と一体化して作動する．センサを通して外界の情報をとらえる．外界の変化に呼応して自身の動作を変化させ，アクチュエータを通して外界に働きかける．すなわち，制御不能な外界の情報を扱うというオープン性がある．同時に，外界の時間経過に対処するため，ソフトウェアを入力と出力を結ぶ変換処理としてとらえるだけでは不十分である．処理にかかる時間や処理の進行過程を明示的に扱う必要がある．

このような点から，組み込みソフトウェアの特徴をよく表す計算モデルとして，状態遷移の考え方を採用することが多い．実際，多くの設計方法論が状態遷移システムをモデリングの道具として採用している[16]．UML ではステートダイアグラムが相当する．

上記に述べた第 1 の特徴である並行性に関しては，1 つの UML ステートダイアグラムの中に並行動作すると解釈可能なコンポーネントを複数持つことができる．第 2 のリアクティブ性については，イベントに対する状態遷移として処理の進行を表現することができる．第 3 の活性については状態遷移の処理進行過程を詳細に調べることができる．これらの特徴を持つので，UML ステートダイアグラムはモデリングの道具として，組み込みソフトウェア開発の場で使われるようになってきている．

2.2 UML ステートダイアグラムの動作規則

UML 仕様書[17] が規定しているステートダイアグラム（以下，UML/STD と略す）の標準動作規則の概要を説明する．

2.2.1 階層型状態遷移システム

UML/STD は D. Harel が提案した Statecharts に基づく階層型状態遷移システムである．ある状態が内部に状態遷移システムを持つという意味で，階層的な状態遷移システムである．階層には 2 種類あり，And 階層と Or 階層と呼ぶ．And 階層として展開される複数のステートマシンは並行実行すると考える．Or 階層内では遷移関係によって状態が移り変わり，いずれか 1 つの状態にある．

図 2 に簡単な例[14] を示した．このシステム TV は Working と Waiting の Or 階層からなり，Working は Image と Sound を構成要素とする And 階層からなる．すなわち，Image と Sound は並行実行する，等を表現している．また，Image は末端状態である Picture と Text を持つ Or 階層状態である．

UML/STD は多様な機能を持つ複雑な言語であり，UML 仕様書では状態遷移の動作規則を自然言語で説明している．本稿では形式的な取扱いを行うために，文献 12) を参考に，ステートダイアグラムの特徴である階層性（And ならびに Or）と Run-to-Completion と呼ぶ動作の考え方を中心に説明する．以下，設計者が記述する表層的な状態遷移システムをステートマシンと呼び，一方，動作規則では次に導入するコンフィギュレーションマシンを参照する．

2.2.2 コンフィギュレーション

UML/STD の実行スナップショットを表現するために，コンフィギュレーション（Configuration）と呼ぶ項表現を導入する．図 2 で，動作開始直後の状況を表す初期状態は

```
TV(Working(Image(Picture),Sound(On)))
```

のような項として表現できる．この表現を用いると，UML/STD の動作規則を，コンフィギュレーションを状態としてイベントが遷移先の状態を決定する状態遷移マシン，すなわちコンフィギュレーションマシンとして定義することができる．

UML/STD での状態遷移は，Configuration 間の遷移として表現する．たとえば，図 2 のいくつかの遷移は

```
TV(Working(Image(Picture),Sound(?X)))
-(txt)->
TV(Working(Image(Text),Sound(?X)))

TV(Working(Image(?X),Sound(On)))
-(mute)->
TV(Working(Image(?X),Sound(Off)))
```

のように表現できる．ここで，`?X` は変数であり don'tcare を表すと考える．上の初期項に対しては，txt と mute とが同時に発火することが可能である．この場合は，2 つの遷移規則を同時に適用することで次の状態に至る．

```
TV(Working(Image(Text),Sound(Off)))
```

また，変数 `?X`，`?Y` を用いることで，イベント out によるレベル間遷移も簡明に記述できる．

```
TV(Working(?X,?Y))
-(out)->
TV(Waiting(Dis-connected))
```

コンフィギュレーション項

X を変数の可算集合，Σ を記号の有限集合とする．また，関数 arity を $e \in \Sigma$ が持つ引数の数とする．

$$\mathrm{arity} : \Sigma \to \mathcal{N}_0$$

$\mathrm{arity}(e) = 0$ であるような Σ^0 は末端の状態を表す．同様に，$\mathrm{arity}(e) = 1$ は Or 階層に展開される状態 Σ^{or} を，$\mathrm{arity}(e) > 1$ は And 階層に展開される状態 Σ^{and} を表すとする．このとき，コンフィギュレーション項は以下である．

$$s := x \mid a \mid v(s) \mid w(s_1, \ldots, s_n)$$

ただし，$x \in X$，$a \in \Sigma^0$，$v \in \Sigma^{or}$，$w \in \Sigma^{and}$ である．

コンフィギュレーション項の全体を $T_\Sigma(X)$ と書く．特に，基底項の集まり $T_\Sigma(\emptyset)$ を T_Σ と書く．

また，補助関数として，$T_\Sigma(X)$ 上のユニフィケーション unify を導入する．

$$\mathrm{unify} : T_\Sigma(X) \times T_\Sigma(X) \to \mathrm{Bool} \times (X \leftrightarrow T_\Sigma(X))$$

便宜上，unify を 2 つに分割して扱う．

$$\mathrm{unify} = <\sigma, \rho>$$
$$\sigma : T_\Sigma(X) \times T_\Sigma(X) \to \mathrm{Bool}$$
$$\rho : T_\Sigma(X) \times T_\Sigma(X) \to (X \leftrightarrow T_\Sigma(X))$$

関数 unify を用いることで 2 つのコンフィギュレーション項に重なりがあるか等を計算することが容易になる．

2.2.3 コンフィギュレーションマシン

次に，コンフィギュレーションマシンを定義する．本稿は，階層型状態遷移システムとしての UML/STD サブセットを考察の対象とする．コンフィギュレーションマシンは，

(State, Event, Rule)

と考えることができる．

状態 $\mathrm{State} \subseteq T_\Sigma$

すなわち，基底のコンフィギュレーション項の集合である．

イベント $\mathrm{Event} \subseteq E_n \cup E_c$

E_n 通常イベント

E_c 終了イベント

終了イベントは当該状態での処理終了による脱出遷移を表すための特別のイベントである．

遷移 $t \in \mathrm{Rule}$ には以下の関数が定義されている．

source : $\mathrm{Rule} \to T_\Sigma(X)$

target : $\mathrm{Rule} \to T_\Sigma(X)$

trigger : $\mathrm{Rule} \to \mathrm{Event}$

effect : $\mathrm{Rule} \to \mathrm{Power(Event)}$

guard : $\mathrm{Rule} \to (W \to \mathrm{Bool})$

source と target は各々遷移元と遷移先の状態を指定するが，基底項ではなく変数を持つこと（$T_\Sigma(X)$）に注意すべきである．なお，W は guard(t) を評価し値を確定するために必要な適当なデータドメインとする．

2.2.4 動作規則

動作規則は，あるイベントに対して発火可能な遷移を求めて実行する方法である．この一連のステップの開始時点ではシステムは

<Configuration, EventSet>

で与えられる平衡状態にあり，動作終了時には新たな平衡状態に達する．平衡状態から平衡状態に至るまで，この一連のステップを間断なく実行することを，Run-To-Completion（RTC）と呼ぶ．すなわち，RTC 単位に処理が進行する．特に，一連の動作過程で生成したイベントは以降の RTC 単位になってはじめて実行される．RTC の動作を**図 3** に示す．ただし，文献 12), 15) 等にならって EventSet を FIFO キューとした☆．

図 3 中，RTC 単位は `while...do...end` の本体である StepRTC() にあたる．現在のコンフィギュレーション項 currentState に対して eventQueue から取り出したイベント currentEvent を評価の対象として発火可能な遷移を求める．このとき，衝突する遷移を取り除く必要がある．以下，図 3 の要点を説明する．

(1) 現在の状態 currentState を遷移元に持つ遷移 active をすべての遷移規則 Rule から求める．

☆ UML の標準規則はイベントプールの管理方法を規定していない．

```
OriginalSimpleRule() =
  while true
  do
     StepRTC()
  end

StepRTC() =
  if not empty(eventQueue)
  then
    currentEvent := dequeue(eventQueue);
    newState := currentState;
    for each t in Gamma
    do
       newState := nextState(t, newState);
       add(eventQueue, executeEffect(t));
    end;
    currentState := newState;
```

図 3 RTC の動作
Fig. 3 RTC execution steps.

active : State → Power(Rule)
active (s) = { t ∈ Rule |
unify(source(t), s) = <true, _> }

(2) active から現在のイベント currentEevnt とデータ w ∈ W に対して発火可能な遷移 enabled を求める．

enabled : State × Event
→ (W → Power(Rule))
enabled (s, e)(w) = { t ∈ Rule |
t ∈ active(s) ∧ e1(t, e)(w) }
e1 : Rule × Event → (W → Bool)
e1 (t, e)(w) = (trigger(t) = e) ∧ guard(t)(w)

(3) enabled から衝突の可能性を考慮して本当に実行できる遷移の集合 Gamma を求める．

(4) 遷移 ∀t ∈ Gamma を発火させて次のコンフィギュレーション状態を決定する．

nextState(t, s) = ∃1 s' ∈ State •
unify(source(t), s) = <true, ρ>
∧ target(t).ρ = s'

ただし，target(t).ρ は，当該遷移 t の遷移先コンフィギュレーション項に，ユニフィケーション結果を代入して遷移先のコンフィギュレーションを具体的に求めることを示す．

次に上記 (3) で導入した Gamma の計算方法を説明する．

(1) active から衝突する可能性のある遷移の集合 conflict を求める．

conflict : State → Power(Rule)
conflict = { t ∈ Rule |
∃ t' ∈ active(s) • c1(t, t') }
c1 : Rule × Rule → Bool
c1 (t1, t2) = (unify(source(t1), source(t2))
= <true, ρ_s>) ∧ $\rho_s \neq \emptyset$
∧ unify(target(t1), target(t2))
= <false, ∅>

t1 と t2 との遷移元は重なりがあるが両立し，したがって，同時に遷移可能となる可能性がある．一方，遷移先には重なりがなく両立しない状態に至ることを示す．すなわち，t1 と t2 が衝突する．

(2) conflict から，優先度を考慮することで実行可能と見なせる復活遷移の集合 promoted を求める．ここで遷移の間の優先度の関係を ≻ とした．

promoted : State → Power(Rule)
promoted(s) = { t ∈ Rule |
¬ (∃ t' ∈ conflict(s) • t' ≻ t) }

(3) 発火可能性のある遷移の集合 fireable を求める．

fireable : State × Event → Power(Rule)
fireable(s, e) = (enabled(s, e) − conflict(s))
∪ promoted(s)

(4) 衝突しない遷移の最も大きな集合 Gamma(s, e) を求める．Gamma(s, e) ⊆ fireable(s, e) は以下の条件を満たす．

∀t, t' ∈ Gamma(s, e) • ¬ c1(t, t')
∧ (t ∈ fireable(s, e) ∧ (∃t' • ¬ c1(t, t'))
⇒ t ∈ Gamma(s, e))

ここで，t ∈ $\bar{\text{f}}$ireable(s, e) の選び方は特に決まっていない．非決定的に選択されると考えるのが UML/STD の標準動作規則である．

上記の定義の中で，遷移のプライオリティを表現する順序関係 ≻ を用いた．UML/STD の標準動作規則では，補助関数 cover を用いて，状態間の順序関係を定義し，これを用いて，遷移のプライオリティ関係を計算する[17]．

t1 ≻ t2 = source(t1) ≻ source(t2)
s1 ≻ s2 = cover(s2, s1) ∧ ¬cover(s1, s2)
cover : $T_\Sigma(X) \times T_\Sigma(X)$ → Bool
cover(a, b) =
∃ <flag, ρ> • (unify(a, b) = <flag, ρ>)
∧ flag ∧ a.ρ = b

直観的には，状態 a が状態 b を部分項として持つスーパステートであることを示す．なお，遷移のプライオリティを表す順序関係 ≻ として，異なる定義を導入することも可能である．上記の UML/STD 標準と異なるプライオリティ関係を用いた Statecharts バリエーションも歴史的には検討された[16]．

上の定義で enabled(currentState, currentEvent)(w) が空集合になる場合がある．このとき，currentEvent は遷移の発火に寄与できない．標準の動作規則では，このイベントは何も効果がなく次の RTC ステップに移る．すなわち，イベントは消滅したことと同等であり，これを，暗黙の消滅（implicit consumption）と呼ぶ[17]．

なお，2 つの遷移 ∀t1, t2 ∈ Gamma に対して実行順序によらず結果が同じになる，すなわち，t1 と t2 の実行に関する可換性が成り立つことを示すことができるので，図 3 にあるように任意の順序で遷移を実行してよい．

最後に，eventQueue として FIFO キューを用いる典型的な方法の場合[12),15)] の終了イベント E_c の取扱いに触れる．遷移の効果として生成された通常イベント E_n は eventQueue の最後尾にキューされる．一方，UML の標準規則では，終了イベント E_c を優先して評価する必要があるとする．これを実現するために eventQueue の先頭にキューする．

2.3 問 題 点

図 1 を参照し，組み込みソフトウェアに適用する場合，標準の動作規則が持つ問題点を考察する．

組み込みソフトウェアはセンサからのイベントに呼応して動作する．実世界が持つ並行性という特徴から複数のセンサが状態検知し，したがって，複数イベントが同時に発生する可能性がある．さらに，外部の実世界は逐一状態を変化させるため，イベントの原因が瞬時に消滅するかもしれない．すなわち，すべてのイベントを確実に取り込む必要がある．

一方，UML 標準の動作規則では，1 つの RTC サイクルでは，1 つのイベントを処理することを想定している．ステートダイアグラムの遷移規則に並行性があって，複数イベントを同時に扱える場合であっても，1 つのイベントに限定する．さらに，通常の解釈では，イベントプールは FIFO キューとして構成する[12),15)]．評価対象とならないイベントは待ち行列に格納される．したがって，発生イベントがどのタイミングで評価対象となるか予測できない．さらに，イベントが評価対象となる時点，すなわち，キューから取り出された時点のコンフィギュレーションによっては遷移に寄与することなく消滅する（implicit consumption）．

一般に，協調して全体の機能を果たす複数の状態遷移マシンが，同一ハードウェアの上で適切に作動するためには，ハードウェアの適切な配分，すなわち，スケジューリングを行わなければならない．また，状態遷移マシンに実行の優先順序がある場合，競合する複数のイベント間に優先順序がある場合等を考慮する必要がある．

このようなスケジューリングに関しては独自の技術が開発されている．しかし，スケジューリングの問題は実行基盤上での動作を制御する技術であり，UML ステートダイアグラムが扱うようなデザイン段階での抽象レベルよりも具体的であることが多い．一方，デザイン段階での振舞い検証では，スケジューリング・アルゴリズムの詳細な違いを議論するよりも，高い抽象レベルでの比較検討が望ましい[6]．

また，最近では車載システムをはじめ組み込みシステムの技術が分散ソフトウェア化している．本稿で考察するデザイン表現の枠組みでは，複数のステートダイアグラムが通信基盤を介して協調動作する状況に相当する．UML の用語を用いると，ステートダイアグラムとコラボレーションの双方を一体化して考える必要があることになる．

一般に，通信を含むコラボレーションを表現するためには，通信の方式を決定しなければならない．通信の送り手と受け手とが同期通信を行う場合，通信メッセージを一時的に保持するバッファを介した非同期通信がある．後者の場合，バッファ長の選択によって振舞いが変わるので注意すべきである．いいかえると，通信を含むコラボレーションを考慮することは，コラボレーションへの参加者であるステートダイアグラムの動作と通信の方式といった 2 種類の動作規則を扱うことになる．

以上，UML ステートダイアグラムを組み込みソフトウェアのデザイン記述に適用する場合の問題点を考察した．イベントの取扱い，スケジューリング，通信コラボレーションの 3 つの観点が欠けていることを指摘した．

通信コラボレーションに関しては，もともと UML/STD だけは議論できない．本稿では，イベントの取扱いとスケジューリングの 2 つに対して，イベントと状態進行に関わるプライオリティ概念の導入という観点から，2.2 節に説明した動作規則を拡張する案を議論する．

3. 組み込みソフトウェアの振舞い仕様検証

本稿では，2.3 節で指摘した問題点を解決するために，プライオリティ概念を表現する記述を導入し標準の動作規則を変更する．

3.1 拡張の方針

第 1 に，基本的な動作規則は前節の RTC サイクルとするが，その時点のイベントプール全体を処理対象

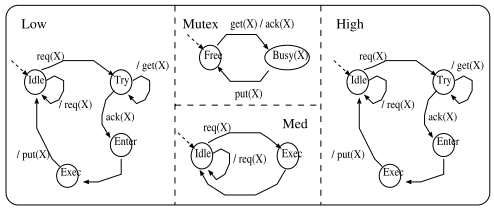

図 4 ステートダイアグラム System
Fig. 4 System.

とする．すなわち，1 つの RTC サイクルで複数イベントを評価対象とする．これによって，並行動作するステートマシンが独立なイベントを「同時に」評価することができる．逆に，複数イベントを同一サイクルで評価するために，イベントが衝突する可能性がある．

第 2 に，イベント衝突時の優先度を決定するために，イベント体系の性質として，順序関係を導入する．たとえば，緊急性の高いイベントと通常イベントが競合する場合，緊急イベントを遷移の評価対象とする，等を表現できる．競合したために遷移の評価対象とならなかったイベントは暗黙の消滅の規則により消える．しかし，消滅させたくないイベントもある．これを次 RTC サイクルで評価することを指定するために，イベントに遅延特性を導入する．

第 3 に，論理的なスケジューリングを行うために，状態遷移の進行を制御するメタレベルの機構を導入する．構文上のステートマシンに対して，状態遷移の進行を許可する条件（`provided` 句）を付加する．たとえば，高い優先度のタスクが実行状態であるとき，低い優先度のタスクは状態遷移進行できない，という優先スケジューリングの考え方を容易に表現できる．

なお，他ステートダイアグラムとの通信コラボレーションを表現するために，各ステートダイアグラムは内部動作に加えて通信機能を有するが通信の方式は別途規定すると考える．本稿の範囲では，外部から通信によって到着するイベントを外部イベントと呼び，デフォールトで遅延特性を与えて，とりこぼされることがないようにする．

図 4 に 3.5 節で詳述する実験で用いる例題のステートダイアグラムを示す．これは，プライオリティに基づくタスクスケジューリングと排他的な共有資源がある場合に起きる可能性のある不具合を扱う例題であり，優先度の概念が本質的な役割を果たす．文献 6) でも取り扱っている例題であり，惑星探索器 Mars Pathfinder の組み込みソフトウェアで実際に発生した不具合である．そのため，提案の記述方法ならびに動作規則の検討例題として有用である．

本稿の方法では，優先度を容易に表現することができる．第 1 に，タスクの優先度の違いを表現するために provided 句を導入する．

$$\text{Med provided } \neg((\text{High in Exec}) \vee (\text{High in Enter}))$$
$$\text{Low provided } \neg((\text{High in Exec}) \vee (\text{High in Enter})) \wedge \neg(\text{Med in Exec})$$

これによって，たとえば，`Med` は，`High` が `Exec` あるいは `Enter` 状態でないときにのみ状態遷移を進行できることを表現する．

さらに，イベントの優先度を用いて，同時競合した場合の優先順位を指定する．

$$\text{req(High)} \succ \text{req(Med)} \succ \text{req(Low)}$$

1 つの RTC サイクルで衝突した場合に，High タスクによる実行要求を優先する等を表現することができる．

以上，明示的にスケジューラを用いる必要がなく，排他的な共有資源ならびにタスクの振舞いを状態遷移に基づく考え方で表現するだけで，今考えている問題を議論できるという長所がある．

なお，文献 6) では不具合を再現することを示すための最小モデルを扱う．一方，上記で議論した本稿のモデル記述は，Mars Pathfinder で不具合が顕在化したタスク状況を素直に反映したものになっている．同じ不具合の状況を議論の対象としているが，モデル化の目的ならびに観点が異なる．

3.2 拡張 UML/STD

本稿で考える UML/STD のサブセットを

$$(\text{State}, \text{Event}, \text{Rule}, \text{ProvidedClauses})$$

で定義する．

ProvidedClauses は階層を構成するステートマシンに対して定義する遷移進行の条件である．

$$\text{N} \in \Sigma^{or}$$

をステートマシンの名前とするとき，次の構文で指定する．

$$\text{N provided P}$$

ここで，P は以下の構文で表現する命題である．

$$\text{P} := \text{M in S} \mid \text{true} \mid \text{false} \mid \neg P \mid P \wedge P \mid P \vee P$$

命題 (M in S) はステートマシン M がコンフィギュレーション S にいるときに true となる．

次に複数イベント間の競合関係を解消するためにイベント系に優先度関係（$\succ$）を定義する．直観的な順序関係を表し，その意味は，後に定義する topMostEvents で与える．

$$\text{Event}^{\succ} = <\text{Event}, \succ>$$

また，イベントごとに遅延評価を行うか否かを指定する遅延特性を与え，各イベントは次の関数を持つと

```
ProposedSimpleRule() =
  while true
  do
    ProposedStepRTC()
  end

ProposedStepRTC() =
  if not empty(eventPool)
  then
    eventSet := topMostEvents(eventPool);
    newState := currentState;
    for each t in Gamma
    do
      if provided(currentState, t, providedClauses)
      then
        newState := nextState(t, newState);
        add(executedRule, t);
        add(eventPool, executeEffect(t));
    end;
    set(eventPool,externalEvents);
    add(eventPool,defferedEvents);
    currentState := newState;
  end
```

図 5 本稿で提案する動作規則の処理流れ
Fig. 5 Proposed execution steps.

する．通信コラボレーションによりシステム外部から到来するメッセージに対応するイベントには遅延特性を与えるとする．

deferred : Event → Bool

以上，本稿では，UML/STD の中核である And/Or 階層状態遷移マシンに，provided 句と順序関係を持つ Event$^{\succ}$ を導入することを提案した．

3.3 動作規則

本稿で提案する動作規則の処理流れを**図 5** に示した．標準の動作規則（図 3）への主な修正・追加は以下の 3 点である．

- イベント集合に保持されているすべてのイベントを同時に評価する．イベント衝突を解消するために関数 topMostEvents を用いる．
- provided 句の条件を満たすステートマシンについてのみ遷移を実行する．条件判定を行う述語 provided を用いる．実際に実行した遷移を集め executedRule とする．
- 通信コラボレーションにより外部から到着したイベント externalEvents ならびに発火に寄与しなかった遅延イベントをイベント集合に取り込む．

次に提案動作規則の主要点を厳密に定義する．第 1 に，1 つの RTC サイクルで複数イベントを評価対象とするように標準の動作規則を変更する．

- 自身よりも高い優先度のイベントが eventPool 内にないイベントだけを集める．

$$\text{topMostEvents}: \text{Power(Event)} \to \text{Power(Event)}$$
$$\text{topMostEvents(es)} = \{\, e \in es \mid \neg(\exists e' \in es \bullet e' \succ e) \,\}$$

- enabled の定義を複数イベントに拡張する．

$$\text{enabled} : \text{State} \times \text{Power(Event)} \to (W \to \text{Power(Rule)})$$
$$\text{enabled}(s, es)(w) = \{\, t \in \text{Rule} \mid t \in \text{active}(s) \wedge (\forall e \in es \bullet e1(t, e)(w)) \,\}$$

第 2 に，provided 句に関係する進行制御を行う必要がある．

- ProvidedClauses は先の構文定義（N provided P）から得ることができ，ステートマシン名と遷移進行条件の対応関係である．
  ProvidedClauses : $\Sigma \leftrightarrow$ Proposition
- 発火可能な遷移の集合 Gamma を決定してから，provided が true になったステートマシンだけを遷移実行させる．同時に，実行した遷移を executedRules に集めておく．

第 3 に，次 RTC サイクルで評価するイベントを集める処理は 3 通りある．

- 遷移にともなう効果（executeEffect(t)）として生成されるイベントの集まり．
- 通信コラボレーションによって外部から到来するメッセージに起因するイベントの集まり ExternalEvents.
- 遷移の発火に寄与しなかったイベントであって遅延特性を持つイベントの集まりは以下の式で 2 つの集合の差として計算する集合．

$$\{\, e \in \text{Event} \mid \forall t \in (\text{Gamma} - \text{executedRules}) \bullet \text{defferd}(\text{trigger}(t)) \,\} - \{\, e \in \text{Event} \mid \forall t \in \text{executedRules} \bullet \text{defferd}(\text{trigger}(t)) \,\}$$

最後に，終了イベント E_c は通常イベントよりも優先度が高いとすればよい．FIFO キューとする場合では先頭にキューする等の特殊な扱いをする必要があったが，本提案方式では，イベントの優先度で統一的に取り扱うことができる．

3.4 検証性質の表現

次に検証性質の表現方法を考えるために，ステートマシンの振舞い仕様を定義する．基本的には，RTC 動作を Configuration の列を生成する規則として，この列を振舞い仕様と考えればよい．

Run の定義

$\pi = s_0 s_1 \ldots s_n \ldots$

ここで，s_{k+1} は s_k から RTC 動作の規則（図 5）で

計算されるコンフィギュレーション項である．さらに，RTC の平衡状態が，

<Configuration, EventPool>

の組で表現されることを考慮すると，検証性質に EventPool を入れておくほうが都合が良い．そこで次の定義を得る．

拡張 Run の定義

$\widehat{\pi} = \langle s_0, \xi_0 \rangle \langle s_1, \xi_1 \rangle \ldots \langle s_n, \xi_n \rangle \ldots$

ここで，s_k は上記と同様であり，ξ_k は RTC 動作の規則で計算される EventPool である．

振舞い仕様の性質表現として，本稿では線形時相論理（Linear Temporal Logic，LTL）の式 f を扱う．LTL の論理式 f は，通常の論理結合子（$\neg$，$\wedge$，$\vee$，$\Rightarrow$）に，3 つの時相オペレータ $\Box$（always），$\Diamond$（eventually），U（strong until）を追加して表現するとする．なお，3 つの時相オペレータの意味は通常の定義[6] と同じなので省略する．

最後にステートマシンの検証方法を考察する．図 3 ならびに図 5 記載の動作規則は実行のための規則である．Gamma の構成方法から分かるように，衝突のない遷移だけを実行させる．一方，一般に衝突しない遷移の集合の選び方は 1 通りに決まらない．標準の動作規則では遷移の間に優先度の概念を導入することで一部の衝突を解消する．優先度により選択した後，さらに衝突があっても，遷移を非決定的に選択することで衝突しない遷移だけからなる Gamma を構成した．

逆に，このことは，Gamma の具体的な構成方法を複数通り考えることが可能であることを示している．検証するという立場では可能なすべての経路について与えられた性質が成り立つか否かを検査する．すなわち，可能な方法で構成したすべての Gamma_j に対して図 5 の RTC 動作規則によって網羅的に生成された拡張 Run の集合 $\{\widehat{\pi}\}$ を得て，この集合の要素に対して与えられた式 f が成り立つか否かを検査すること（$\widehat{\pi} \models \text{f}$）である．

3.5 Promela/SPIN への変換

本稿ではモデル検査ツール SPIN[6] を用いて振舞い検証を行う．基本的には，動作規則を反映する Configuration マシンを構成し，遷移集合の計算処理を含むイベント評価ならびに RTC を実現する動作規則をモデル検査ツールの入力仕様言語 Promela で表現すればよい．網羅的な経路生成を行うためには，非決定的な遷移計算を行うように Promela 記述を生成すればよい．

図 5 の動作規則では，図 3 と同様に，Gamma の計算を `currentEvent` と `currentState` から行っている．Gamma 計算をモデル検査の過程でインタプリタ的に行うと探索する状態空間が非常に大きくなりモデル検査に基づく検証の効率が大幅に低下する．一方，Gamma の計算は与えられたステートダイアグラムに対してあらかじめ計算してから Promela 記述を生成するセミコンパイル方式が考えられる[11),14),15)]．本稿でも同様な方法を採用した．

```
if
 :: (stateTV == Working) ->
   if
    :: eventOff -> stateTV = Waiting;
       stateWaiting = StandBy; nullWorking()
    :: eventOut  -> stateTV = Waiting;
       stateWaiting = Disconnected; nullWorking()
    :: else ->
       if
        :: (stateImage == Picture) ->
           if
            :: eventTxt -> stateImage = Text
            :: else -> skip
           fi
        :: (stateImage == Text) ->
           if
             :: eventTxt -> stateImage = Picture
             :: else -> skip
           fi
        :: else -> skip
       fi;
  ...
fi
```

図 6 Promela 記述の断片
Fig. 6 A Fragment of Promela.

図 6 は冒頭に示した例（図 2）に対する Configuration マシンを Promela で表現した記述の断片である．ここで，`stateTV`，`stateWaiting`，`stateImage` といったブール変数を導入して，Configration を指定する方法を採用した．たとえば，ブール値の式

```
stateTV == Working
&& stateImage == Picture
&& stateSound == On
```

は，コンフィギュレーション

```
TV(Working(Image(Picture),Sound(On)))
```

を表す．これらのブール変数はコンフィギュレーションを直接的に表現するので `provided` 句の計算が容易になる．たとえば，命題 `(Image in Picture)` は次の論理式に変換することができる．

```
stateImage == Picture .
```

図 6 において，`if...fi` の各条件分岐に，発火すべき遷移の条件を書き表している．条件式 `(stateTV == Working)` に対応する分岐には状態 `Working` を遷移元とする遷移をまとめた．

条件分岐 eventOff は図 2 の off 遷移を表す．ここで，eventOff はブール値の変数であって，当該イベントが EventPool 中に存在するときに *true* となる．

図 2 から分かるように，遷移実行後，状態 Working から Waiting のサブ状態 StandBy に移る．Promela 記述のうえでは，代入文 stateWaiting = StandBy で表現している．なお，nullWorking() は状態 Working ならびにそのサブ状態を表現する変数をすべて無効化する処理である．以上によって，Working から Waiting に遷移したことを表現する．

3.6 SPIN を用いる検証実験

3.1 節ならびに図 4 で説明した例題の振舞い検証を行う．プライオリティに基づくタスクスケジューリングと排他的な共有資源がある場合に起きる可能性のある不具合である「プライオリティ逆転」を扱う例題である．

複数のタスクが同時に実行要求する場合，優先度を考慮して実行スケジュールする．High，Med，Low の順番とする．この例では，High と Low とが資源 Mutex を共有する場合を考えている．特に，Low が Mutex を獲得して，High が資源解放待ちであって，さらに，Med が実行要求するときを考える．Med と Low の優先度比較によって，Med に実行権が与えられる．Med が占有実行すると，Low は実行されない．Low は共有資源 Mutex をロックしているため，High は資源解放待ちになり動作しない．すなわち，Med が実行し High が実行できないという優先度が逆転した状況に陥る．

まず，図 4 のステートダイアグラムをコンフィギュレーションの方法で定式化し，Promela で表現する．次に，本システム全体に対して，性質を時相論理 LTL の式として表現して検証する．以下，興味ある性質の検証結果を示す．

$$\Box((\text{High in Try}) \rightarrow \Diamond(\text{High in Exec}))$$

この式は進行性，あるいは leads-to の性質を表現するもので，High は共有資源獲得要求（Try 状態）の後，いつか作動状態（Exec）に移行することを示す．正しく共有資源を獲得する場合，この式は満たされる．しかし，本例の記述では，中間優先度の Med が作動することによって，Low が共有資源を保有したままの状態にいるという状況が反例となる．High よりも優先度の低い Med が実行することを示し，したがって，プライオリティ逆転の現象を生じることが分かる．たとえば，Low が共有資源をとらないように図 4 のモデル記述を変更すると，上記の進行性はつねに成り立つ．

同様に，Low の振舞いを以下の進行性の式を用いて解析する．

$$\Box((\text{Low in Try}) \rightarrow \Diamond(\text{Low in Exec}))$$

High の場合と同様に，Low が共有資源を保有した後に，優先度の高い Med が作動するために，Low の動作が進行しないという状況が反例となる．当然であるが，Med を作動させない場合には満たされる．

最後に，Med がつねに動作進行するか否かを調べる．

$$\Box((\text{Med in Idle}) \wedge \text{req(Med)} \rightarrow \Diamond(\text{Med in Exec}))$$

この式の反例としては，Med と High の実行要求（req）が衝突し，上に定義したイベントの順序関係に従って，High の要求が受け付けられるシナリオを得る．Med が飢餓状態になる可能性があることを示し，現実のシステムでは，適切な公平性を導入する必要があることを示唆する．この式は，イベントの存在（req(Med)）を命題に含み，拡張 Run $\widehat{\pi}$ を導入することの必要性を示す．

Promela への変換に際しては，本例題が，複雑な階層構造を持たないことを利用して，状態数を削減するように工夫した．特に，この例題は，図 2 が示すような階層レベル間の遷移がないために，And 階層の構成要素であるステートマシンを単純に組み合わせることで全体を表現できるという性質を利用した．状態空間の大きさを調べるために不具合を発生しない場合を測定すると 8,000 状態足らずであった．このように階層構造が単純であることを利用して状態空間を削減する方法は重要である．扱った例題では直観的にあきらかであるが，一般的には自明ではない．削減を可能とする条件を見つけることは今後の課題である．

4. 関 連 研 究

UML ステートダイアグラムについて，Harel の Statecharts 以降，多くのバリエーションが提案されている[5),16)]．標準の仕様書で規定する動作規則はおおむね確定したようである[17)]．1 つ大切なことは，Rhapsody での動作規則[5)] を含めて，これらは実行可能とすることを目的とした動作規則であるという点である．特に，実行可能なプログラム生成を目的としており，検証を行うための意味論ではない．

UML 標準の動作規則は自然言語で説明されていて曖昧であるため厳密に形式化する必要である．本稿でも採用した Configuration マシンの書き換えによる方法[12)] が標準的な手法になっている．動作規則の形式化は，形式検証を行う基礎として避けて通れない[1),9),11),14),15)]．これらの研究において，UML/STD のモデル検査という問題の共通点は，本稿で議論した階層型状態遷移システムと RTC ステップに代表される動作規則を扱うことである．UML/STD が持つほ

かの多様な機能についての取扱い範囲は異なる．本稿の方法は文献 12) をベースとし，イベントと状態進行に関わるプライオリティ概念を標準の動作規則に統合した点が新規である．

HUGO[15] は Promela への変換方法を工夫し，半コンパイルと呼ぶ事前評価・最適化の手法によって素朴な方法に比べて状態数を 3 桁少なくできることを示した．本稿の簡単な実験でも Promela での表現方法が状態数に大きな影響を与えることを確認した．

文献 4) は，要求仕様の段階での検証に適用することを想定して，UML/STD 標準規則とは異なる解釈を与えている．特に，1 つの RTC ステップで複数イベントを処理する点が本稿の方法と同じである．要求仕様と組み込みソフトウェアといった異なる観点の研究であるが，同じ考え方が有効であるとしている．UML/STD で欠けている重要な拡張の方法と考えられる．

イベントに優先順位を導入する考え方はプロセス代数で提案されている．文献 2) は割込み等の緊急性の高いアクションを表現するために優先度の考え方を導入した．文献 13) は 1 つのデザイン記述に対して複数の遷移間の優先度を変えることで，異なる観点からのデザイン検証を行うという stress testing の基本機構として優先度を導入した．なお，UML/STD[17] や文献 13) が扱う遷移の競合は，非決定性オートマトンの非決定的な遷移に相当し，イベント競合ではない．本稿の方法は，優先度の順序関係を持つイベント体系を導入し，さらに，文献 2) と同様に複数のイベントが発火遷移に影響する場合を含む．

UML/STD の通常の考え方では，状態遷移の進行制御を行うために，遷移にガード条件を与える方法を用いる．本稿で扱うスケジューリングの側面を遷移ガードで表現しようとすると，複数の遷移に条件を付与しなければならない．スケジューリング条件が複数箇所に散らばるため，書き落とし等の誤りが混入する可能性が大きい．一方，provided 句の方法はステートマシンに対して与える条件であるため見通しが良い．

Promela/SPIN[6] では，provided 句に，任意の条件を書くことができる．書き方によっては，provided 句の評価時に状態爆発を起こす．柔軟性が高い反面，使いにくい．本稿の方法は，論理的なスケジューリングとしての状態遷移の進行制御に用いることに専用化することで意味と使い方を明確にした．特に，命題をうまく選ぶことで，評価時に状態爆発が起こらないようにしている．

最後に，UML の世界では，RT UML と呼ぶスケジューリング・性能・時間に関するプロファイルが提案されている[3]．しかし，RT UML は，実時間特性を持つシステムのデザインを表現する基本概念・基本語彙をオブジェクト指向概念で整理した体系であり，UML/STD の動作規則にスケジューリング概念を導入するものではない．

5. おわりに

UML ステートダイアグラムを組み込みソフトウェアのデザイン記述に適用する場合の問題点を考察し，イベントの取扱いとスケジューリングの観点を中心に考察した．

UML ステートダイアグラムの標準動作規則を変更して，組み込みソフトウェアが持つ特性を容易に表現する新たな規則を提案した．特に，コンフィギュレーションマシンの考え方を採用して動作規則を厳密に定義した．さらに，提案の規則に従って動作するコンフィギュレーションマシンを Promela に変換することで，モデル検査ツール SPIN を用いた性質検証が可能であることを具体例で示した．

今後，より規模の大きな例題を対象とした検討を進めることで，本提案方式を実用的な技術に発展させていきたい．

謝辞 本研究の一部は文部科学省リーディングプロジェクト「基盤ソフトウェアの総合開発」の「高信頼性組込みソフトウェア構築技術」[7),8)] として，北陸先端科学技術大学院大学からの再委託研究として実施した．片山卓也教授ほかプロジェクトメンバに感謝する．

参考文献

1) Clarke, E. and Heinle, W.: Modular Translation of Statecharts to SMV, CMU-CS-00-XXX, (Aug. 2000).
2) Cleaveland, R. and Hennessy, M.: Priorities in Process Algebras, *Information and Computation*, No.87, pp.58–77 (1990).
3) Douglass, B.P.: *Real Time UML*, 3rd Edition, Addison Wesley (2004).
4) Eshuis, R., Jansen, D.N. and Wieringa, R.: Requirements-Level Semantics and Model Checking of Object-Oriented Statecharts, *Requirements Eng.*, pp.7:243–263 (2002).
5) Harel, D. and Kugler, H.: The Rhapsody Semantics of Statecharts, LNCS 3147, pp.325–354, Springer-Verlag (2004).
6) Holzmann, G.: *The SPIN Model Checker*, Addison-Wesley (2004).
7) 片山卓也：最新ソフトウェア技術による高信頼性組込みソフトウェアの開発，*IPA SEC Journal*,

No.1, pp.8–15 (Jan. 2005).
8) Katayama, T., et al.: Highly Reliable Embedded Software Development Using Advanced Software Technologies, *IEICE Trans. Infor. Syst.*, Vol.E88-D, No.6, pp.1105–1116, (June 2005).
9) Lattela, D., Majzik, I. and Massink, M.: Automatic Verification of a Behavioural Subset of UML Statechart Diagrams using the SPIN Model-Checker (1999).
10) Lee, E.A.: Embedded Software, *Advances in Computers*, Vol.56, Academic Press (2002).
11) Lilius, J. and Paltor, I.P.: vUML: a Tool for Verifying UML Models, *TUCS TR*, No.272 (May 1999).
12) Lilius, J. and Paltor, I.P.: The Semantics of UML State Machines, *TUCS TR*, No.273 (May 1999).
13) Magee, J. and Kramer, J.: *Concurrency — State Model & Java Programs*, Wiley (1999).
14) Mikk, E., Lakhnech, Y., Siegel, M. and Holzmann, G.: Implementing Statecharts in Promela/SPIN, *Proc. WIFT'98* (1998).
15) Schafer, T., Knapp, A. and Merz, S.: Model Checking UML Stata Machines and Collaborations, *Electronic Notes in Theoretical Computer Science*, Vol.55, No.3 (2001).
16) Wieringa, R.J.: *Design Methods for Reactive Systems*, Morgan Kaufmann (2003).
17) OMG — Unified Modeling Language, v.1.5, (Mar. 2003).

(平成 17 年 3 月 2 日受付)
(平成 17 年 9 月 2 日採録)

中島　震（正会員）

1981 年東京大学大学院理学系研究科修士課程修了．同年 NEC 入社．同社 R&D グループ，法政大学を経て，2004 年情報・システム研究機構国立情報学研究所教授．2005 年から総合研究大学院大学教授を併任．この間，1988〜1989 年米国オレゴン大学客員研究員．2001 年から科学技術振興機構 PRESTO「機能と構成」領域研究員，ついで 2004 年から SORST 研究員を兼務．2004 年から北陸先端科学技術大学院大学 JJREX 客員教授を兼務．ソフトウェアの形式仕様と検証に関する研究に従事．学術博士（東京大学）．2002 年度山下記念研究賞，2003 年度日本ソフトウェア科学会論文賞受賞．日本ソフトウェア科学会，ACM 各会員．