4K-06

DEVSモデルとプロセス指向モデルの混合シミュレーション手法

櫻井　拓也†　　井上　潮†

東京電機大学 工学部情報通信工学科†

1. はじめに

離散事象システムのモデル化には，さまざまな手法(world view)や形式(formalism)が存在し，システムの特性によって，適切な手法が異なる．しかし，複数の手法でモデル化したシステムをシミュレーションしようとすると，シミュレータの実装や，モデル間の通信方法の調整に問題が生じる．そこで，さまざまな手法でモデル化されたシステムを，統一的な形式で表すことを考える．

離散事象システムの形式のひとつに *DEVS* (*Discrete Event System Specification*) がある．DEVSはさまざまな形式から変換できることが報告されており[2]，統一的な形式として十分な表現力をもっていると考えられる．

しかし，モデル化手法のひとつであるプロセス指向を用いてモデル化したシステムを，DEVS へ自動的に変換する方法については，著者が知る限り発表されていない．

本稿では，プロセス指向のモデルをDEVSに自動的に変換する手法を提案する．また，そのシミュレータをF#で実装したことを報告する．

2. DEVS

DEVSモデルは，次の組で表される[1].

$$M = (X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta)$$

X : 入力イベントの集合．Y : 出力イベントの集合.
S : 状態の集合.
$\delta_{ext}: Q \times X \to S$: 外部(external)遷移関数.
　ただし，$Q = \{(s, e) | s \in S, 0 \le e \le ta(s)\}$,
　　e は前回の状態遷移からの経過時間.
$\delta_{int}: S \to S$: 内部(internal)遷移関数.
$\lambda: S \to Y$: 出力関数.
$ta: S \to \mathbb{R}_0^+ \cup \infty$: 時間経過関数.

ある状態 s のとき，$ta(s)$ だけ時間が経過したら $\lambda(s)$ を出力し，次の状態 $\delta_{int}(s)$ に遷移する．もし，$ta(s)$ だけ時間が経過する前に入力イベントがあれば，$\delta_{ext}(s, e, x)$ に遷移する．

3. DEVSモデルへの変換手法

プロセス指向で記述されたモデルを，DEVS モデルに変換する方法を以下に説明する．本手法では，大きく次のように変換を行う．

1. 手続きを，時間を経過させたり，外部イベントを待機したりといった，時間が不連続になる処理がある点で分割する．これをノードとした，制御フローグラフのようなものを，*プロセスグラフ*と呼ぶことにする．
2. 1)次に実行するノード番号，2)手続き中の変数，3)待機処理の内容，4)時刻，5)入出力イベントのキュー，を DEVS モデルの状態とする．遷移関数では，ノードを実行し，状態を更新する．

3.1. プロセスグラフ

プロセスグラフのノードは，処理を行い，次に実行すべきノードへの辺のラベルと，*遷移条件* (どのような状態になるまで待機するか)を返すプログラムを持つ.

手続き中のイベント待機処理部で，待機を行う前までの処理のノードと，待機を行った後の処理のノードに分割する．もし，イベント待機処理が if や while といった制御構文のブロック内にあるなら，条件が真，偽それぞれのときに実行する処理を含むノードに分割する(図 1).

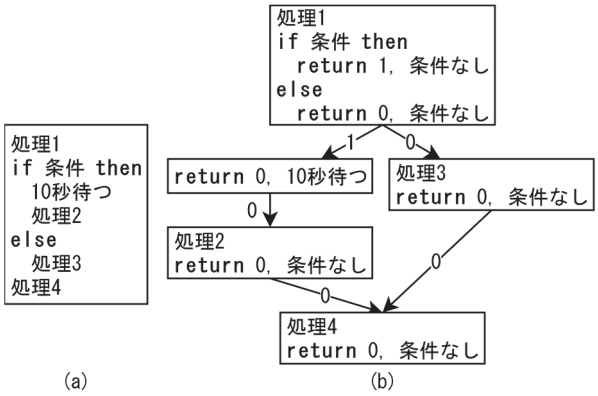


図 1: ifブロック内に待機があるときのノード分割．(a)元の手続き．(b)(a)を分割した結果.

3.2. モデルの状態と入出力

生成するDEVSモデルの状態は，次の組で表される構造を持つ.

$$s = (nidx, vars, cond, t, in, out) \in S$$

$nidx \in N$: 次に実行するノード番号，または END.
$vars \in \mathrm{V}$: 変数の配列．$cond \in C$: 遷移条件.
t : 現在の時刻．in : 入力イベントキュー.
out : 出力イベントキュー.

An integrated simulation method of DEVS and process-oriented models.
Sakurai Takuya, Inoue Ushio
†Dept. of Information & Communication Engineering, Tokyo Denki University

Copyright ©2020 Information Processing Society of Japan.
All Rights Reserved.

3.3. モデルを構成する関数

プロセスグラフのi番目のノードが持つプログラムを，変数の配列とイベント待機結果(R)を入力し，出力イベント，次のノード番号，更新された変数の配列，遷移条件を出力する関数 $G_i: V \times R \to Y \times N \times V \times C$ として表すことにする．

このとき，DEVS シミュレータでシミュレーションできるように，アルゴリズム 1 に示す関数 δ' を用いて，内部遷移関数 δ_{int}，外部遷移関数 δ_{ext} をアルゴリズム 2，アルゴリズム 3 に示すように与える．ただし，$cta: S \to \mathbb{R}_0^+ \cup \infty$ は遷移条件から時間経過を求める関数，$wres: S \to S \times R$ はイベント待機結果を求める関数とする．また，出力関数，時間遷移関数を次のように与える．

$$\lambda(s) = \begin{cases} peek(s.out) & s.out \text{ is not empty} \\ \emptyset & s.out \text{ is empty} \end{cases}$$

$$ta(s) = \begin{cases} 0 & s.out \text{ is not empty} \\ +\infty & s.nidx = END \\ cta(s) & \text{otherwise} \end{cases}$$

アルゴリズム 1: 状態遷移関数 $\delta'(s)$

if $cta(s) > 0$ **then return** s
$(s', r) \leftarrow wres(s)$
if $s'.nidx = END$ **then return** s'
$(out', nidx', vars', cond') \leftarrow G_{s'.nidx}(s'.vars, r)$
return $\delta'\big((nidx', vars', cond', s'.in, out')\big)$

アルゴリズム 2: 内部遷移関数 $\delta_{int}(s)$

$out' \leftarrow$ **dequeue** $s.out$ **if** $s.out$ is not empty
$s' \leftarrow (s.nidx, s.vars, s.cond, s.t + e, s.in, out')$
if $s'.out$ is empty **then return** $\delta'(s')$ **else return** s'

アルゴリズム 3: 外部遷移関数 $\delta_{ext}(s, e, x)$

$in' \leftarrow$ **enqueue** x **to** $s.in$
$s' \leftarrow (s.nidx, s.vars, s.cond, s.t + e, in', s.out)$
if $s'.out$ is empty **then return** $\delta'(s')$ **else return** s'

4. F#による実装

本手法によるシミュレーションを，汎用プログラミング言語に組み込めることを確認するため，DEVS シミュレータと，プロセス指向でモデルを記述するためのライブラリを F#で実装した．ライブラリは，F#のコンピュテーション式を用いて手続きを記述すると，その抽象構文木から，上記のアルゴリズムで DEVS に変換する．

4.1. 変換例

具体例として，複数台の機械と修理人が働いているシステムを考える．機械は故障が多く，修理人が修理しながら稼働している．ここで，機械が連続して稼働する平均時間(MTBF)と，機械が故障して修理にかかる平均時間(MTTR)が設定されているとき，各機械の挙動は，次のように表せる．

1. 平均が MTBF と等しい時間だけ連続して稼働する．
2. 故障する．
3. 修理人を呼び，到着するまで待つ．
4. 平均が MTTR と等しい時間をかけて，修理する．
5. 修理人を返す．
6. 1 へ戻る．

リスト 1 に，このモデルを F#で記述した例を示す．また，プロセスグラフへの変換結果を図 2 に示す．このモデルと，修理人リソースを表すモデルとの結合(coupled)モデルを作成することで，シミュレーションを行うことができた．

リスト 1: F#で記述したモデルの例

```
type MachineEvent = | Up | Down
                    | Repairing | Repaired
let machine id seed =
  (fun p -> processModel {
    let rng = ref seed
    while true do
      emitOutput Up p //稼働する
      do! wait (ttf rng) p
      emitOutput Down p //故障する
      //修理人を待つ
      let! _ = receiveEvent anyEvent p
      emitOutput Repairing p
      do! wait (ttr rng) p //修理する
      emitOutput Repaired p //修理人を返す
  }) |> ProcessModel.createAtomicModel
```

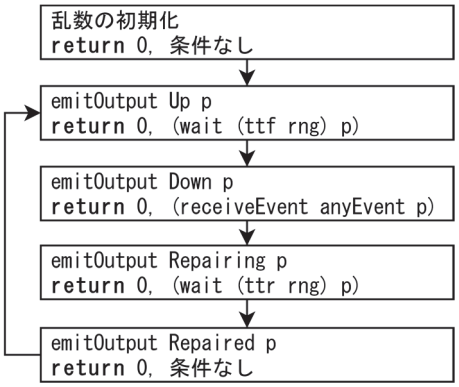


図 2: リスト 1 のプロセスグラフ

5. おわりに

手続き的に記述されたプロセス指向モデルを，DEVS モデルに自動的に変換する方法を提案し，F#でシミュレータを実装した．これにより，プロセス指向でモデル化を行ったシステムを含む，複雑なシステムのモデル化に，DEVS を利用できることを確認した．

今後の課題として，動的にプロセスを作成するようなモデルの記述方法や，シミュレーションを並列実行するための拡張方法の検討が必要と考える．

参考文献

[1] Zeigler, B. P., Muzy, A. and Kofman, E.: *Theory of Modeling and Simulation 3rd Edition*, p.155, Academic Press (2018).
[2] Vangheluwe, H: DEVS as a common denominator for multi-formalism hybrid systems modelling, *Proc. CACSD*, IEEE (2000).

Copyright ©2020 Information Processing Society of Japan.
All Rights Reserved.