Vol.2013-SE-179 No.31
2013/3/12

実時間システム向けの文脈指向DSL

中村 遼太郎[1,a)]　渡部 卓雄[2,b)]

概要：本研究の目的は適応的実時間システムのためのドメイン固有言語 (DSL) の提案である．組込みシステム等に要求される性質である適応性と実時間性はいずれも横断的関心事であり，プログラムを複雑化する要因となる．適応的な振舞は実行時の環境を表す情報（文脈）に依存する動作として記述できる．そしてそのような記述のモジュール化を促進する手法として文脈指向プログラミング (COP) が提案されている．我々は，時刻および時区間を文脈とみなすことで，COP の考え方が適応的実時間システムに有効であることを示している．従来，COP を支援する機構は言語処理系を拡張して導入されることが一般的であったが，本研究ではプログラミング言語 Scala の諸機能を用いて COP のための内部 DSL を実現することで，言語処理系を拡張することなく適応的実時間システムのモジュラーな記述が可能になることを示す．

キーワード：文脈指向プログラミング, 実時間システム, ドメイン固有言語, Scala

1. はじめに

1.1 文脈指向プログラミング

プログラム実行時の様々な環境や状況を表す情報を文脈 **(context)** と呼ぶ．例えば携帯端末等の組込みシステムの場合，位置情報，電源の状況，利用可能な通信手段といった外部環境に関する情報は文脈である．また，実行状態を陽に表すデータやユーザインターフェースの状態など，プログラム実行中に得られる各種内部情報も文脈とみなすことができる．文脈やその変化に依存する振舞いを示すシステムは適応的 **(adaptive)** あるいは文脈依存 **(context dependent)** であると言われる．組込みシステムは多くの場合文脈依存であることが求められる．

文脈はいわゆる横断的関心事であり，文脈依存の振舞いを条件文などでナイーブに記述すると，それらがプログラム中に散逸してしまうことがある．そこで，そのような記述のモジュール化を促進するプログラミングパラダイムとして文脈指向プログラミング **(Context-Oriented Programming, COP)** が提案されている [8]．

現在の主な COP の方式では，文脈依存な振舞いをモジュール化する単位である層 **(layer)** と呼ばれる概念が用いられている．文脈の変化に応じて層を活性化および非活性化させること（以下，活性化制御とする）で文脈依存の振舞いを表現でき，結果として適応的なプログラムのモジュール性を高めている．

層およびその活性化制御は，プログラミング言語の新たな構文として導入されるのが一般的であり（例えば ContextJ[2], JCop[3], ContextL[5], Subjective-C[6] など），具体的にはブロックやメソッド等の構文単位と連動させる構文が用いられる．そのようにすることで活性化制御に関する制御の流れを限定し，意図しない振舞同士の衝突などの問題を起こりにくくしている．一方，外部要因（例えばセンサの値の変化）やプログラム中の指定された実行ポイント（例えばメソッド呼出し）をイベントとみなし，それらをトリガーとする柔軟な活性化制御方式を持つ言語 EventCJ[9] も提案されている．この方式では，イベント発生時における層の活性化制御は層遷移規則として宣言的に記述される．

1.2 文脈としての時間

前節で述べたように組込みシステムの多くは文脈依存であるが，それに加えて実時間性が要件に加えられることがある．そのようなシステム，すなわち適応的実時間システムでは，文脈に依存する記述と時間制約に関する記述が混在し，プログラムを複雑化する要因となる．これは時間制約に関する記述がプログラムの随所に現れるためである．このことは時間制約も横断的関心事であることを示している．

そこで我々は時刻および時区間を文脈と見なすことで，文脈指向プログラミングの考え方が実時間システムの記述

[1] 東京工業大学・工学部情報工学科
[2] 東京工業大学・大学院情報理工学研究科・計算工学専攻
Department of Computer Science, Tokyo Institute of Technology
[a)] ryotaro@psg.cs.titech.ac.jp
[b)] takuo@acm.org

に有効であることを示し，この考え方にもとづいて設計した文脈指向プログラミング言語 ProcneJ を提案・実装した [14], [15]．時間はプログラムの実行とは独立に経過するため，位置情報などの外部要因による文脈の変化と同様にとらえることができる．ProcneJ では時間の経過をイベントとして扱い，EventCJ のような層遷移規則を用いて文脈の活性化制御を行っている．時間に関する記述は層遷移規則を含むプロセス記述というモジュールに限定され，それによって時間制約に関する記述のモジュール化が可能になっている．さらにプロセス記述から時間オートマトン [1] の記述を生成することで，時間制約や層の活性化に関する性質の UPPAAL[10] による検証を可能にしている．

1.3 ドメイン固有言語による実現

第 1.1 節で述べたように，現在までの文脈指向プログラミングの提案は，層やその活性化制御に関する機構を既存の言語処理系を拡張することで導入しているものが多い．ProcneJ も Java に新しい構文を導入した言語であり，その処理系は Java および AspectJ への変換系という形で実現されている．

このように新たな言語や言語機構（特に新しい構文）の導入は，文脈依存の部分を明確にモジュール化できるという利点を持つが，その一方で既存の開発ツールやフレームワーク等が利用できなくことにつながる．例えば，文脈指向プログラミングのための導入した構文を使用したファイルの編集時は，元の構文を想定した補完機能を利用することができない．

そこで本研究で我々はプログラミング言語 Scala の諸機能を用いて文脈指向プログラミングと実時間プログラミングのための内部 DSL を実現する手法を提案する．それにより，言語処理系を拡張することなく適応的実時間システムのモジュラーな記述が可能になることを示す．例題の記述を通して提案手法の有効性を示す．また，新たな構文を導入した場合との比較を行う．

以下，第 2 節で我々が提案する Scala 上の DSL とその実現方式について述べ，第 3 節でその評価を行う．第 4 節で関連研究にふれ，第 5 で結論と今後の課題について述べる．

2. 提案する DSL とその実現方式

2.1 例題

本説では，提案手法を説明するために用いる例題について説明する．ここでは簡単なランプを例とし，その動作を時間オートマトン [1] を用いて記述する．図 **1** がランプの動作を表す時間オートマトンである．時間オートマトンは有限オートマトン[*1]に時計変数および時計変数を含むガードと不変式を導入したものであり，実時間制約に依存する

[*1] 正確には Büchi オートマトン

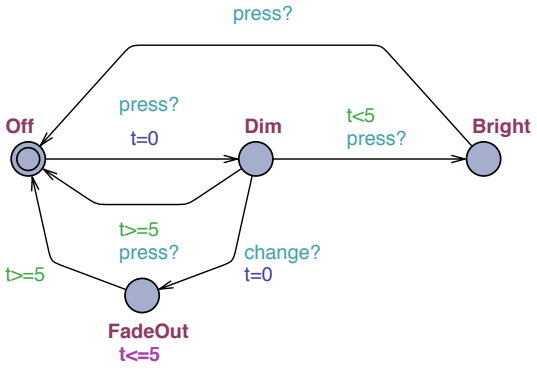

図 1 ランプのモデル

動作を記述することができる．図 1 では t が時計変数である．時計変数は一定の割合で増加する．状態 Dim から状態 Bright および状態 Off への遷移にそれぞれ t<5 および t >= 5 という式が指定されているが，これらはガードであり，遷移するためにはそれらを満たさなければならない．

図 1 のランプは，消灯 (Off) 時に press ボタンを押すと音が鳴り薄暗く点灯 (Dim) する．Dim の状態になってから，5 単位時間経過する前に再度 press ボタンを押すと明るく点灯 (Bright) し，5 単位時間以上経過した後で press ボタンを押下すると消灯する．Dim の状態で change ボタンを押すと，フェードアウトを始め (FadeOut)，5 単位時間かけて少しずつ暗くなり，やがて消灯する．Bright の状態で press ボタンが押下されると消灯する．

2.2 層の表現

提案手法では層の表現に内部クラスを利用し，その中に文脈依存の振舞いを記述する．トレイト Layer をミックスインした内部クラスが層となる．トレイト Layer を内部で宣言するクラスは TimedActor をミックスインしなければならない．TimedActor は Actor をミックスインしたアクターである．図 **2** では，press ボタンが押下されたときのメソッド press の振舞いを，Off と Dim の状態の場合に分けて定義している．

提案手法では，TimedActor をミックスインしたオブジェクト（以下，文脈依存のオジェクト）が自分の活性状態の層を管理する．活性化した層を配列 ActiveLayers の先頭に挿入することで，活性化後の経過時間が短い順に，先頭から層を activeLayers に整列させる．

2.3 層に定義した振舞い

提案手法がサポートする文脈に依存した振舞いの最小単位はメソッドである．TimedActor で定義するメソッド layered, apply() は最後に活性化した層を返す．これにより，クラス内部に書かれた式 layered.press() と，クラス外部に書かれた式 lamp().press() は，文脈依存のオブジェクト lamp の最後に活性化した層で定義された

```
1  object Lamp extends TimedActor {
2    trait LampState extends Layer {
3      ..
4    }
5    object Off extends LampState {
6      def press () = {..}
7      def change() = {..}
8    }
9    object Dim extends LampState {
10     def press () = {..}
11     def change() = {..}
12   }
13     ..
14 }
```

図 2　層の構成

```
1  trait TimedActor extends Actor {
2    val activeLayers =
3            new ArrayBuffer[Layer]
4    ...
5  }
```

図 3　活性状態の層をまとめる配列

メソッド press を呼び出せる．例えば，Off が活性，Dim が非活性状態である場合，図 2 の 3 行目の層 Off で定義したメソッド press が呼び出される．

activeLayers の先頭要素は最後に活性化した層を指すので，layered と apply() はどちらも activeLayers のを先頭要素を返している．なお，式 lamp() は式 lamp.apply() を省略した形式である．Scala では，オブジェクトに続く括弧内に 0 個以上の引数を収めてオブジェクトに適用すると，そのオブジェクトのメソッド apply の呼出し式に変換される．

```
1  def layered = activeLayers(0)
2  def apply() = layered
```

activeLayers の要素の型は Layer であり，また，トレイト Layer はメソッド press を定義していないので，本来は式 lamp().press() はエラーとなる．提案手法は，メソッド layred や apply をもちいた層のメソッド呼び出し時の型エラーを，暗黙の型変換によって解決する．暗黙の型変換は，コンパイラが型の不一致を検出したときに，コンパイラが暗黙的に型エラー検出箇所と同じスコープ内にある implicit とついたメソッドを適用することで，型エラーの解消を試みる機能である．ランプの例では，コンパイラは，図 4 の 2 行目で宣言されたメソッド conv を，メソッド layered, apply を通して層のメソッドを出すときに適用し，型エラーを回避する．これにより，

```
1  object Lamp extends TimedActor {
2    implicit def conv[T<: Layer](l: T)
3      = l.asInstanceOf[LampState]
4
5    trait LampState extends Layer {
6      def press()
7      def change(){}
8    }
9    object Off extends LampState {
10   ..
11 }
```

図 4　暗黙の型変換

```
1  object Off extends LampState {
2    def press() {
3      proceed.press()
4      ..
5    }
6    ..
7  }
```

図 5　proceed の使用例

lamp().asInstanceOf[LampState].press() のように逐一型のキャストメソッドを挟む手間を省くことができる．

提案手法は，同時に複数の層を活性状態にすることができる．層の中で使用するメソッド proceed は呼出し元の層の次に活性期間が短い層を返す．例えば，図 5 の 3 行目に proceed.press() は，次に活性期間が短い層のメソッド press を呼び出す．

メソッド proceed は，層自身を activeLayers に渡し，自身のインデックス番号を把握することで，次に活性期間が短い層を見つけ出す．本来，proceed を呼び出すためには，層自身をパラメーターとして渡さなければならない．しかし，暗黙のパラメーターを図 6 の 1 行目に宣言しているので，層自身を渡さずに，proceed を呼び出すことができる．暗黙のパラメーターは，メソッドに渡たされた引数の数が不足しているときに，コンパイラが同じスコープにある implicit とついた同名の変数を引数に加え，呼出し式を完成させる機能である．暗黙のパラメーターを宣言することで，proceed(this) のように自身を明示的に proceed に渡すことなく，proceed を呼び出すことができる．

2.4　層の活性化，非活性化時の処理

層の中には，ブロック activate{..}, deactivate{..} を記述できる．層の活性化，非

```
implicit val myself: Layer = this
def proceed(implicit myself: Layer)
                                  : Layer {
  val index = activeLayers indexOf myself
  ..
  activeLayers(index + 1)
}
```

図 6　proceed の実装

```
  def activate(f: => Unit) {
    acti = () => f
  }
  def deactivate (f: => Unit) {
    deacti = () => f
  }
```

図 7　activate, deactivate の実装

活性化時に，これらのブロックが評価される．ブロック activate は，クラスのコンストラクターと同様に，メソッドに必要な初期化処理を行う役割を果たす．ブロック deactivate は，オブジェクトの状態を，その層が活性化する前の状態に戻す処理の記述に利用できる．

activate, deactivate の実体は，無引数の関数をパラメーターにとる高階関数である．図 **7** の f は名前渡しパラメーターである．名前渡しパラメーターには，空の関数パラメーターを省いた関数本体だけ記述することで無引数の関数を渡すことができる．

2.5　層の遷移記述

applyRule メソッドに Rule オブジェクトを渡すことで，層の活性状態を変えることができる．Rule は層の活性状況を切替える条件と遷移先の層をクラスパラメーターにもつケースクラスである．表 **1** に Rule の構造を示す．Rule の構文と意味は EventCJ の層遷移規則を模倣している．表 1 の *ID* は層を指す．applyRule は渡たされた Rule を調べ，その結果によって表 **2** の通りに層を遷移する．

Rule	::=	Rule(*Condition*, *Operator*(*NewContext*))
Condition	::=	*ID* | Not(*ID*) | ⋆
Operator	::=	SwitchTo | Activate
NewContext	::=	*ID* | __

表 **1**　Rule の構造

Layer や表 1 の Not, SwitchTo, Activate, ⋆, __は図 **8** のようにミックスインされているので，Rule はコンパイラによる構文チェックを受ける．例えば，Rule(__, SwitchTo(⋆)) は，コンパイラの型チェックを通らない．

Rule(⋆, SwitchTo(__))	全てのレイヤーを非活性化する
Rule(⋆, *Operator*(*l*))	*l* を活性化する
Rule(Not(l_1), *Operator*(l_2))	l_1 が非活性状態ならば l_2 を活性化する
Rule(*l*, SwitchTo(__))	*l* を非活性化する
Rule(l_1, SwitchTo(l_2))	l_1 が活性状態ならば、l_1 を非活性化し、l_2 を活性化する
Rule(l_1, Activate(l_2))	l_1 が活性状態ならば l_2 を活性化する

表 **2**　Rule の規則

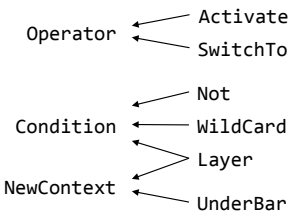

図 8　Rule のミックスイン

```
object Dim extends LampState {
  val t = Clock(0)
  activate  {
    t <-- 0
  }
  def press() {
    if (t >= 5) {
      applyRule(Rule(Low, SwitchTo(Off)))
    } else {
      applyRule(Rule(Low,
                        SwitchTo(Bright)))
    }
  }
  ..
}
```

図 9　Clock の使用例

2.6　時間変数

ケースクラス Clock は，ある時点からの経過時間を秒単位で記録する．Clock は，メソッド<--による代入，四則演算，大小比較のメソッドを備えている．図 **9** に Clock の使用例をあげる．ここでは，ランプが Dim の状態になってからの経過時間を計測するために Clock を使用している．ランプの状態が Dim に遷移した直後に，4 行目の式 t<--0 は，t が記録する経過時間を 0 にリセットする．また，t は，経過時間に基づく press ボタン押下後の遷移先を決める式 t>=5 の中で使用されている．

Clock は，スレッドセーフな，弱い参照のキーをもつ clockMap のキーである (図 **10**)．オブジェクト Clock のメソッドは，キーに対応する値である Int 型のオブジェ

```
1 val clockMap =
2         new WeakHashMap[Clock, Int]
3         with SynchronizedMap[Clock, Int]
```

```
1 case class Clock(init: Int) {
2   def clocks = ClockCounter.clockMap
3   def >= (t: Int) = clocks(this) >= t
4   ..
5 }
```

図 10　Clock の実装

```
1 object FadeOut extends LampState {
2   activate {
3     println("Activate FadeOut")
4     aperiodicTask(5000)
5     {applyRule(Rule(FadeOut,
6                     SwitchTo(Off)))}
7   }
8   ..
9 }
```

図 11　aperiodicTask の使用例

クトに引数を渡し，同名のメソッドを呼び出した結果を返している．例えば，図 10 の 3 行目に定義したメソッド >=に渡たされた Int 型のオブジェクトは，その Clock に対応するマップの値 clocks(this) のメソッド>=に渡される．また，アクター ClockCounter が，1 秒ごとに clockMap が管理する各値を 1 ずつ足している．

2.7　周期的，非周期的な振舞い

periodicTask, aperiodicTask に渡たされた式は，一緒に渡した整数オブジェクトが示す時間間隔，あるいは，時間後に評価される．はじめの引数に時間を表わす整数を渡し，次の引数に後で評価したい式を渡す．aperiodicTask の使用例を図 **11** に示す．ここでは，図 1 の 5 秒間のフェードアウト後にランプの状態を Off に戻すために，4-6 行目で aperiodicTask を利用している．

periodicTask, aperiodicTask に渡した式は，アクターに渡され，そのアクター上で後で評価される．図 **12** は periodicTask の実装である．式を評価するアクターは，指定した time 間隔で自分自身に TIMEOUT を送信する．TIMEOUT メッセージを受信したアクターは式を評価し，その後 TIMEOUT の受信まで待機する．

3.　提案手法の評価

3.1　他の手法による例題の実装との比較

この節では，提案手法を評価する．評価には図 1 のランプのモデルを利用する．提案手法を利用しないランプの実装と ProcneJ を利用したランプ実装を用意し，それらを提案手法と比較する．

```
1 def periodicTask(time: Long)
2                 (f: => Any): Actor = {
3   def ptask {
4     Actor.reactWithin(time) {
5       case TIMEOUT => f; ptask
6       case STOP =>
7     }
8   }
9   Actor.actor(ptask)
10 }
```

図 12　periodicTask の実装

3.1.1　提案手法を利用しない実装との比較

図 **13** は，提案手法を利用しない図 1 の実装の一部である．2 行目の変数 lampState が，現在のランプの状態 (Off や Dim など) を値にもつことで，現在の状態を管理する．ランプの文脈に依存する振舞い (press や change) を，ランプの状態を表わすクラスに定義する．6 行目のメソッド changeState によって lampState が参照するオブジェクトを変更することで，メソッド press や change の振舞いを動的に変更する．

提案手法の proceed は super と異なり，それが指し示すオブジェクトを動的に変えることができる．proceed は自身の次に活性期間の短い層を返すので，各層の活性状態を変更することで，proceed の振舞いを変えることができる．proceed と super は，press!と標準出力に出力するときの処理に利用される．press!は，Off の状態で press ボタンが押されると表示される．図 13 の実装では，press!の表示を，17 行目の super.press をもちいて実装している．提案手法では図 5 の 3 行目のように proceed.press によって実装している．例えば，ランプの仕様が変わり，消音機能が有効なときは press!と表示しない機能が追加された場合，図 13 の実装では，LampState か Off の press メソッドを変更する必要があるが，提案手法による実装では，Off に加えて何も処理しないメソッド press を定義した層を活性化させれば，既存のメソッドを変更しなくても，振舞いを変えることができる．

提案手法による実装は，図 13 の実装よりも，文脈の切替え時における処理の見通しがよい．提案手法では，図 11 のように，層の活性化時の振舞いが activate ブロックの中に記述されるので，activate 内を確認すれば，文脈が切替わった直後の振舞いがわかる．図 13 では，ランプの状態の切替え時の処理は基本コンストラクタに記述されている．基本コンストラクタは，クラス内部に直接書かれるのでクラス全体を見渡さなければ，初期化時の振舞い全

体を把握することができない．

提案手法による実装では，時間に依存する振舞いを簡潔に記述できる．図 13 の実装では，28 行目のように時刻の差を求めることで，経過時間を調べている．求めた経過時間は，Dim からの遷移先の決定や，FadeOut から Off へ遷移するタイミングを計るために使用される．本手法では，Dim からの遷移先の決定には，図 9 のように Clock オブジェクトを利用し，FadeOut を抜けるタイミングには，図 11 のように aperiodicTask を利用している．これらは差や除算が含まれず簡潔に記述されている．

3.1.2 ProcneJ による実装との比較

ProcneJ のコードは，クラス記述とプロセス記述に分かれる．クラス記述に文脈依存の振舞いを記述し，プロセス記述に時間制約，層を切替えるタイミング，活性化・非活性化する層を記述する．

ProcneJ では，層の遷移規則をプロセス記述に宣言的に記述できるので，層の活性状態を容易に確認することができる．図 13 や提案手法による実装では，文脈に依存する振舞いの中にランプの状態や層を切替える処理 (changeState や applyRule) が埋め込まれているので，コードから実行中の各層の活性状況を読み取ることが難しい．

ProcneJ におけるメソッド proceed は，proceed を利用しているメソッドと同名のメソッドを呼び出す．図 **15** の中の proceed は，Lamp のメソッド press を呼び出している．提案手法の proceed は，別の層を返すメソッドであるから，例えばメソッド press の中で他の層のメソッド press を呼出したい場合に proceed.press と記述する必要があるため，提案手法の層のメソッドを呼び出す式は，ProcneJ のものよりも冗長になる．

3.2 提案手法と ProcneJ の機能の比較

提案手法は ProcneJ のように文脈依存の振舞いと，層の遷移規則を別に記述することができない．また，層の遷移に関するモデル検査のコードを自動生成できない．提案手法の層を切替えるメソッド applyRule は層のメソッド内やコンストラクタ内に記述される．層によってモジュール化をはかる文脈指向言語では，活性状態の層の組合せが変わることで，呼び出されるメソッドが変化する．したがって，実行中の層の活性状況を把握していなければ，期待する層のメソッドを呼び出すことができない．ProcneJ では，文脈依存の振舞いと層の遷移規則を別に記述するので，層の活性状況を確認しやすい．また，プロセス記述から UPPAAL のコードを自動生成することで，層が期待通りに活性化するか確認することができる．

提案手法はメソッドの修飾子 before, after をサポートしない．これらがついたメソッドは，同名のメソッドが実行される前後に実行される．

```
1  object Lamp {
2    var lampState: LampState = new Off
3    def press() {lampState.press()}
4    def change() {lampState.change()}
5
6    def changeState(s: LampState) {
7      lampState = s
8    }
9    trait LampState {
10     def press() {println("press!")}
11     def change() {}
12   }
13   class Off extends LampState {
14     println("Activate Off")
15     override def press() {
16       println("Press in Off")
17       super.press()
18       println("Deactivate Off")
19       changeState(new Dim)
20     }
21   }
22   class Dim extends LampState {
23     val time = currentTimeMillis()
24     println("Activate Dim")
25     override def press() {
26       println("Press in Dim")
27       println("Deactivate Dim")
28       if(((currentTimeMillis() - time)
29                              /1000)>= 5) {
30         changeState(new Off)
31       } else {
32         changeState(new Bright)
33       }
34     }
35     ..
36   }
37   class FadeOut extends LampState {
38     println("Activate FadeOut")
39     val time = currentTimeMillis()
40     while(((currentTimeMillis() -
41                         time)/1000)<= 5){}
42     println("Deactivate FadeOut")
43     changeState(new Off)
44   }
45   ..
46 }
```

図 **13** 提案手法を利用しない実装の一部

```
1              Activate Off
2              Press 'p' or 'c':
3              p
4              Press in Off
5              press!
6              Deactivate Off
7              Activate Dim
```

図 14　ランプのモデルの実行例

```
1  public class Lamp {
2    public void press() {
3      System.out.println("press!");
4    }
5    layer Off {
6      activate {
7        System.out.println("Activate␣Off");
8      }
9      deactivate {
10       System.out.println("Deactivate␣Off"
             );
11     }
12     public void press() {
13       System.out.println("Press␣in␣Off");
14       proceed();
15     }
16   }
17   ..
18 }
```

図 15　ProcneJ のクラス記述の一部

```
1  process Lamp {
2    event press  : * Lamp.press();
3    event change : * Lamp.change();
4
5    init  layer Off {
6      transition : press? then Low;
7    }
8
9    layer Low {
10     transition :
11       after(5) press? then Off;
12     transition :
13       before(4) press? then Bright;
14     transition :
15       change? then FadeOut;
16   }
17
18   layer FadeOut {
19     within [4,5] {
20       then Off;
21     }
22   }
23
24   layer Bright {
25     transition : press? then Off;
26   }
27 }
```

図 16　ProcneJ のプロセス記述

4. 関連研究

実時間処理を表現する言語機構を備えたプログラミング言語はいくつか提案されている．例えば RT-Java[4] では，いくつかの実時間スレッドや実時間処理向けのメモリ管理機構，実時間イベント処理機構やタイマを提供している．実際に時間制約に関する動作はイベントやタイマを用いて記述することになるが，これらは実時間処理のための一般的なプリミティブであり，ProcneJ や本研究で提案しているようなモジュール化機構は提供していない．また抽象度が低いため，記述したコードの検証は困難である．

実時間システムの開発にはモデルベース手法が用いられることがあり，モデルの記述には UML の実時間拡張や Stateflow チャートなどが多く用いられる．その一方で，時間オートマトン [1] に代表される形式的なモデルも用いられている．形式的モデルを用いる利点はモデルの性質についての形式的（かつ機械的）な検証が検証が可能なことである．例えば UPPAAL[10] で検証されたモデル（時間オートマトン）から実行可能なコードを生成する手法がいくつか提案されている [7], [11], [12]．しかし時間オートマトン自体はモジュール化について考慮されているわけではないため，これを直接モデリングに使う場合，システムの規模が大きくなるとモデルが複雑になり，またインクリメンタルにモデルを構築するのも難しいという欠点がある．

さらに，モデルベース開発において，モデルとコードのセマンティックギャップが大きい場合，生成されたコードと手書き（あるいは他の方法で作られた）コードの混用が難しくなる．また，一般にコード生成系を用いる場合はデバッグ（特に低レベルデバッグ）の手間は増加する．プログラミング言語がより抽象度の高い，あるいはモデルに近い記述を支援する場合は，そのような問題は小さくなる．

言語処理系を拡張せずに文脈指向プログラミングの機構を導入する方法として，Java 上のライブラリとして実現した ContextJ*がある [8]．このライブラリの COP に関する機能は限定的であり，記述も煩雑である．

Objective-C には実行時のメソッド探索に失敗したときに

呼び出される `forwardInvocation` という特別なメソッドがある．これを使って言語処理系を拡張することなく COP の機能を提供するものとして ContextFramework[13] がある．この手法では，メソッド探索を失敗させ，レイヤ名を付加した名前をもつメソッドを呼び出すことで文脈依存の動作を実現している．そのためメソッドの名前付けに制約が生じ，また実行時オーバーヘッドが高くなる．

5. 結論と課題

本研究では，適応的実時間プログラミングのための内部ドメイン固有言語を Scala で実装した．例題の実装や他の文脈指向言語の機能との比較を通して提案手法を評価し，言語処理系を拡張して新たな構文を導入することなく，文脈依存の振る舞いと時間依存の振る舞いをモジュール化できることを示した．

提案手法の課題のひとつは，文脈依存の振舞いと層の遷移規則の定義を分離することである．3.2 節で述べたように，層をもちいた文脈指向言語では，活性状態の層の組合せによって，メソッドの振舞いが変化する．そのため，層の活性状況を把握しやすい構文を提供することが文脈指向言語の設計において要点になる．提案手法による実装では，State パターンと同様，振舞いの定義と振舞いを切替える処理が混在するので，層の活性状態をコードから読み取ることが難しい．これらの解決は今後の課題である．

参考文献

[1] Alur, R. and Dill, D. L.: A Theory of Timed Automata, *Theoretical Computer Science*, 126(2), pp. 183–235, (1994).
[2] Appeltauer, M.，Hirschfeld, R.，Haupt, M.，増原英彦：ContextJ: Java 上の文脈指向プログラミング，コンピュータソフトウェア， 28(1), pp. 272–292, (2011).
[3] Appeltauer, M., Hirschfeld, R., Masuhara, H., Haupt, M. and Kawauchi, K.: Event-Specific Software Composition in Context-Oriented Programming, *Software Composition*, LNCS 6144, Springer, pp. 50–65, (2010).
[4] Bollella, M. *et al.*: *The Real-Time Specification for Java*, Addison-Wesley, (2000).
[5] Costanza, P. and Hirschfeld, R.: Language Constructs for Context-oriented Programming: An Overview of ContextL, DLS 2005, ACM, pp. 1–10, (2005).
[6] González, S., *et al.*: Subjective-C: Bringing context to mobile platform programming. *Software Language Engineering (SLE 2010)*, LNCS 6563, pp. 246–265, (2010).
[7] Hakimipour, N., Strooper, P. and Wellings, A.: A model-based development approach for the verification of real-time Java code, *Concurrenty and Computation: Practice and Experience*, 23(13), pp. 1583—1606, (2011).
[8] Hirschfeld, R., Costanza, P. and Nierstrasz, O.: Context-oriented Programming, *Journal of Object Technology*, 7(3), pp. 125–151, (2008).
[9] Kamina, T., Aotani, T. and Masuhara, H.: EventCJ: A Context-Oriented Programming Language with Declarative Event-based Context Transition, AOSD 2011, ACM, pp. 253–264, (2011).
[10] Larsen, K. G., Pettersson, P. and Yi, W.: UPPAAL in a nutshell, *International Journal on Software Tools for Technology Transfer*, 1(1–2), pp. 134–152, (1997).
[11] Pajic, M., Jiang, Z., Lee, I., Sokolsky, O. and Mangharam, R.: From Verification to Implementation: A Model Translation Tool and a Pacemaker Case Study, RTAS 2012, IEEE, (2012).
[12] Senthooran, I. and Watanabe, T.: A Model-Based Approach to Constructing Safe Soft Real-Time Programs for Non-Real-Time Environments, SNPD 2012, ACIS, IEEE Press,pp. 269–274, (2012).
[13] 鈴木将哉, 渡部卓雄: Objective-C による文脈指向プログラミングの実現手法, 電子情報通信学会ソフトウェアサイエンス研究会, SS2012-32, pp. 133-138, (2012).
[14] 安原由貴：組込みシステム向け文脈指向言語の研究，修士論文，東京工業大学大学院情報理工学研究科 (2013).
[15] 安原由貴，森口草介，渡部卓雄：組込みシステムのための文脈指向仕様記述に向けて，日本ソフトウェア科学会第 29 回大会 (2012).