論文 / 著書情報
Article / Book Information

論題(和文)	実時間システム向け文脈指向言語ProcneJ
Title(English)	ProcneJ: A Context-oriented Programming Language for Real-time Systems
著者(和文)	安原由貴, 森口草介, 渡部卓雄
Authors(English)	Yuki Yasuhara, Sosuke Moriguchi, Takuo Watanabe
出典(和文)	, , ,
Citation(English)	, , ,
発行日 / Pub. date	2013, 2
権利情報 / Copyright	

実時間システム向け文脈指向言語ProcneJ

安原 由貴[1,a)]　森口 草介[1,b)]　渡部 卓雄[1,c)]

概要：組込みシステム等において典型的に要求される性質に実時間制約がある．実時間制約は横断的関心事のひとつであり，プログラムを複雑にする．本発表では，文脈指向プログラミングの考え方にもとづいた実時間制約の記述方式とそのモジュール化手法，およびそれにもとづくプログラミング言語 ProcneJ を提案する．ProcneJ では，状態遷移を宣言するプロセス記述と実行可能なコードを記述するクラス記述を分離することで，実時間制約に関する記述のモジュール化を可能にしている．さらにプロセス記述から時間オートマトンを生成することで，時間制約や層の活性化に関する性質の UPPAAL による検証を可能にしている．

キーワード：実時間システム，文脈指向プログラミング

ProcneJ: A Context-Oriented Programming Language for Real-Time Systems

Yuki Yasuhara[1,a)]　Sosuke Moriguchi[1,b)]　Takuo Watanabe[1,c)]

Abstract: Real-time constraints are required typically in embedded systems. They can be regarded as cross-cutting concerns and thus make real-time programs bulky. The goal of this work is to propose a description method of real-time constraints and their modularization technique using the idea of context-oriented programming. In our language ProcneJ, separating the descriptions of event-based state transitions and class definitions enables modular development of real-time programs. In addition, by generating timed automata from the state transition parts, we can verify the consistency of real-time constraints using UPPAAL model checker. In the presentation, we will present the basic idea of our method and will show how it is beneficial for describing real-time systems through examples.

Keywords: Real-Time Systems, Context-Oriented Programming

1. はじめに

プログラムの実行時の様々な環境や状況を表す情報を文脈（context）と呼ぶ．例えば携帯端末上で動作するプログラムの場合，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は文脈の例である．また，状態を陽に表すデータやユーザインタフェースの状態など，一般にプログラム実行中に得られる内部情報も文脈とみなすことができる．

[1] 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)] yasuhara@psg.cs.titech.ac.jp
[b)] chiguri@psg.cs.titech.ac.jp
[c)] takuo@psg.cs.titech.ac.jp

文脈に依存する振る舞いのモジュール化を促進するプログラミングパラダイムを文脈指向プログラミング（Context-Oriented Programming, COP）[1] と呼び，COP を支援する機構を持つ言語を文脈指向プログラミング言語と呼ぶ．現在までに提案されている幾つかの文脈指向プログラミング言語では，文脈依存な振る舞いをまとめる単位である層（layer）と呼ばれる概念が導入されている．

文脈はいわゆる横断的関心事であり，文脈に依存する振る舞いを条件文などで記述した場合，同じ文脈に依存する部分の記述がプログラム中に散在してしまうが，層を用いることでこれらの記述をまとめることができる．そして，文脈の変化に応じて層を活性化/非活性化させることで，文脈依存の振る舞いを表現できる．このような COP の考え

を利用することにより，文脈の認識や切り替え処理に関する部分と実際に行いたい処理の分離が可能になり，プログラムのモジュール性を高めることができる．

層に基づく文脈指向プログラミング言語では，層の活性化と非活性化の制御（以下，活性化制御とする）をブロックやメソッド呼び出しと連動させることで制御フローを限定し，意図しない振る舞い同士の衝突などの問題を起こりにくくしている．さらに，状態遷移モデルを導入することでより柔軟な活性化制御を可能にする文脈指向プログラミング言語 EventCJ[2] も提案されている．EventCJ では，プログラム中の指定された実行ポイントがイベントとみなされ，イベント発生時における層の活性化/非活性化は層遷移規則として宣言的に記述される．また層遷移規則から SPIN モデル検査器用のモデルを生成し，活性化制御に関する性質を検証可能にしている．

時間制約を満たすように構成されたシステムは実時間システムと呼ばれる．実時間システムの開発にはモデルベース開発の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 チャートが用いられることが多い．その一方で，時間オートマトンに代表される形式的なモデルが用いられることもある．形式的モデルを用いる利点はモデルの満たすべき性質についての形式的な検証が可能なことである．例えば，UPPAAL[3] 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．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につれて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（あるいは他の手法で作られた）コードと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．

本発表の目的は，新しい時間制約の記述方式とそのモジュール化の手法，およびそれに基づく言語の提案である．実時間システムのプログラムは，外部環境などの変化に加え，時刻および時間経過に依存する振る舞いを示すプログラムと考えることができる．我々はこの考えに基づいて設計した実時間システム向け文脈指向プログラミング言語 ProcneJ を提案する．ProcneJ では，状態遷移を宣言するプロセス記述と実行可能なコードを記述するクラス記述を分離して記述することで，時間制約に関する記述のモジュール化を可能にしている．さらにプロセス記述から時間オートマトンを生成することで，時間制約や層の活性化

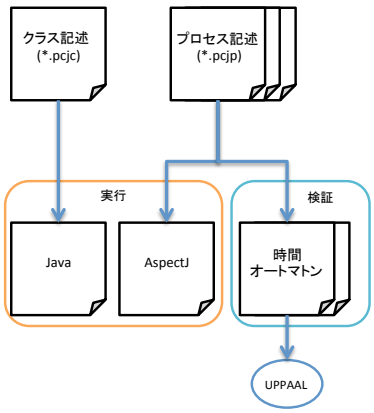

図 1 ProcneJ の構成
Fig. 1 Construction of ProcneJ

に関する性質の UPPAAL による検証を可能にしている．また提案手法により，実行可能なコードの抽象度を高め，モデルにより近い形で記述することができ，モデルとコードのギャップの縮小を可能にする．本発表では，ProcneJ の基本的なアイデアについて説明し，例題を通して文脈指向の考え方による実時間制約記述の有効性について議論する．

以下，第 2 節では ProcneJ の概要をデスクランプの例題を用いて説明する．第 3 節で ProcneJ の実装に関して議論し，第 4 節ではシンプルなトースターの実装の例題を通して ProcneJ の有用性を示す．第 5 節および，第 6 節で関連研究および今後の課題について述べ，第 7 節でまとめを行う．

2. 実時間システム向け文脈指向言語 ProcneJ

本発表で提案する ProcneJ には大きく分けて二つのモジュールが存在する．二つのモジュールをそれぞれ，状態遷移系に相当するプロセス記述と実際の挙動を表現するクラス記述と呼ぶ．**図 1** に ProcneJ の構成を示す．クラス記述は拡張子.pcjc，プロセス記述は拡張子.pcjp を付けたファイルにそれぞれ記述する．一つのプロセス記述は一つのクラス記述内で定義されている層の遷移規則を記述することができる．付録にクラス記述（**図 A·2**）とプロセス記述（**図 A·3**）のそれぞれの構文を載せる．

これらの記述ファイルは ProcneJ のトランスレータによって，クラス記述は標準の Java へ，プロセス記述は AspectJ[4] と時間オートマトンに変換される．生成された Java のコードと AspectJ のコードを組み合わせることにより，プロセス記述の仕様に合わせてクラス記述内の層の活性化制御が行われプログラムが動作する．また，プロセス記述から生成された時間オートマトンは UPPAAL 上で時間制約や層の活性化制御の検証を行うことができる．

クラス記述では，文脈毎の振る舞いを層と部分メソッドを使用して表現している．プロセス記述内ではプロセスを

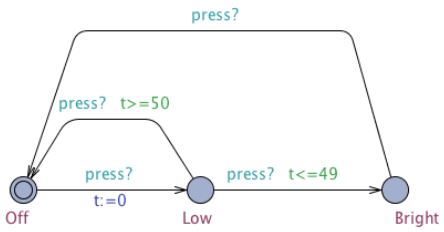

図 2 デスクランプ Ver.1 の時間オートマトン
Fig. 2 Timed automaton of DeskLamp Ver.1

```
1  process Lamp1 {
2     event press: * Lamp.press();
3     init layer Off{
4        transition: press? then Low;
5     }
6     layer Low{
7        transition: after(50) press? then Off;
8        transition: before(49) press? then Bright;
9     }
10    layer Bright{
11       transition: press? then Off;
12    }
13 }
```

図 3 デスクランプ Ver.1 のプロセス記述
Fig. 3 Process description of DeskLamp Ver.1

記述する．プロセスにはクラス記述に対する各層の遷移規則を定義する．同じプロセス内で記述された層が同時に活性化されることはない．また，プロセス記述内にはテストプロセスを記述することも可能である．テストプロセスは検証を補佐する役割を果たす．テストプロセスはトランスレータによって AspectJ へは変換されず，時間オートマトンにのみ変換される．テストプロセスについての詳細は第 4 章で述べる．

2.1 デスクランプ Ver.1

シンプルなデスクランプのシステムを用いて ProcneJ の概要を説明する．このデスクランプはユーザがボタンを押すタイミングによって，ランプが消えている状態から暗く点いている状態か，または明るく点いている状態へと遷移するようなシステムを考える．このシステムをデスクランプ Ver.1 と呼び，時間オートマトンを用いて表すと**図 2** のようになる．

時間オートマトンではシステムの各状態はロケーションと呼ばれる円で表される．各ロケーションにはそれぞれの状態を表す `Off`，`Low`，`Bright` が名付けられている．各ロケーション同士を結ぶ線を辺と呼び，ある状態から他の状態への遷移を表す．辺には幾つかの注釈を付けることが可能であり，それにより他の状態へと遷移するための条件を指定することができる．

例えば，`Off` から `Low` への辺には `press?` と `t:=0` という注釈が付けられている．`press?` は Synchronisation である．プロセス同士はチャンネルを介して同期することが可能であり，Synchronisation は同期イベントの送受信を意味する．`press` はイベント名であり，`press?` は `press` イベントの受信を意味する．一方，`t:=0` は Update である．Update は状態が遷移する際に評価される式を意味する．つまりこの辺には，他のプロセスから送信された `press` イベントを受信した場合に変数 `t` に 0 が代入されることを意味する．

変数 `t` はクロック変数として定義されている．クロック変数は時間の経過とともに一定の割合で値が増えていく変数である．ここでは，`Off` から `Low` へと遷移する際にクロック変数 `t` が 0 になる．`Low` の状態へと遷移した瞬間から `t` の値は時間経過とともに増加し続ける．

`Low` はそれぞれ `Bright` と `Off` へと遷移する二つの辺を持っている．どちらの辺にも Synchronisation と Guard が付いている．Guard は指定されたロケーションへと遷移するための条件を指定する．`Low` から `Bright` への辺には `t<=49` という Guard が付けられている．前述したクロック変数の特性と合わせると，これは `Low` へと遷移してから 49 単位時間以内に `press` イベントを受信した場合に `Bright` へと遷移することを意味する．また，`Low` へと遷移してから 50 単位時間以上経過している場合にイベント `press` を受信した場合には `Off` へと遷移することを意味する．

上述した時間オートマトンをプロセス記述を用いて表すと**図 3** のようになる．この `Lamp1` プロセスから ProcneJ のトランスレータにより，図 2 と同様の時間オートマトンが得られる．

2 行目で `press` イベントとそれに対応する Lamp クラスのメンバメソッドを宣言している．3 行目以降はそれぞれ `Off`，`Low`，`Bright` の各層を定義している．特に層 `Off` の宣言の前には修飾子 `init` がついている．これは層 `Off` が対応するクラス（ここでは Lamp クラス）がインスタンス化されると同時に活性化される層であることを表す．

4 行目では層 `Off` の遷移規則を `transition` を用いて定義している．遷移規則は時間オートマトンにおける辺に相当する．この遷移規則は，`press` イベントを受信した時（`Lamp.press()` が呼びだされた時）に自身の層を非活性化し，`Low` を活性化させることを意味する．

6〜9 行目では層 `Low` の遷移規則を定義している．7 行目も `Off` の遷移規則と同様にイベント `press` を受信した時に `Low` から `Off` へと遷移することを示しているが，この遷移には時間制約が条件としてついている．`transition` を使用して定義した遷移規則は，`after` と `before` を使用して時間制約に関する条件を持つことができる．7 行目

```
1  public class Lamp{
2     private long activateTime;
3     public void print(String str){
4        System.out.println(str);
5     }
6     public void press(){print("Base␣Process");}
7     public long currentTime(){
8        return System.currentTimeMillis();
9     }
10    layer Off{
11       activate{print("Activate␣Off");}
12       deactivate{print("Deactivate␣Off");}
13       public void press(){
14          print("Press␣in␣Off");
15          proceed();
16       }
17    }
18    layer Low{
19       activate{
20          print("Activate␣Low");
21          activateTime=currentTime();
22       }
23       deactivate{print("Deactivate␣Low");}
24       public void press(){
25          print("Press␣in␣Low");
26          print((currentTime()-activateTime)+"␣ms");
27       }
28    }
29    layer Bright{
30       activate{print("Activate␣Bright");}
31       deactivate{print("Deactivate␣Bright");}
32       public void press(){
33          print("Press␣in␣Bright");
34       }
35    }
36 }
```

図 4 デスクランプ Ver.1 のクラス記述
Fig. 4 Description of DeskLamp Ver.1

では Low に遷移してから（層 Low が活性化されてから）50 単位時間以降に press イベントが起きた時に Off へと遷移することを，after(50) を使用して定義している．8 行目では Low に遷移してから 49 単位時間以前にイベント press を受信した時に Bright へと遷移することを，before(49) を使用して定義している．

図 3 のプロセスに対応するクラス記述を**図 4** に示す．同じ文脈に依存する各振る舞いを layer ブロック内で記述する．activate ブロックでは層の活性化と共に実行される振る舞いを定義する．deactivate ブロックでは層の非活性化と共に実行される振舞いを定義する．layer ブロック内では，文脈に依存する各振る舞いを部分メソッドによって表現する．

実行例（**図 5**，**図 6**）と共に見ていく．まず，Lamp クラスをインスタンス化すると，層 Off が活性化される．press メソッドを呼び出す（p を入力する）と，Off が非活性化され，Low が活性化される．また，Off 内で定義された press メソッド（13〜16 行目）は 15 行目で proceed メソッドを呼び出している．これは同時に活性化されている層内で定義されている同名メソッドが呼びだされる．ここでは，Off 以外に活性化されている層はないので，層の外で定義されている press メソッド（6 行目）の振る舞いが呼びだされる．

```
Activate Off
Press 'p' : p
Press in Off
Common Process
Deactivate Off
Activate Low
Press 'p' : p
Press in Low
2431 ms
Deactivate Low
Activate Bright
Press 'p' : p
Press in Bright
Deactivate Bright
Activate Off
```

図 5 素早く 2 回押した時の出力結果
Fig. 5 Result of DeskLamp Ver.1 (Quickly)

```
Activate Off
Press 'p' : p
Press in Off
Common Process
Deactivate Off
Activate Low
Press 'p' : p
Press in Low
8360 ms
Deactivate Low
Activate Off
```

図 6 ゆっくり 2 回押した時の出力結果
Fig. 6 Result of DeskLamp Ver.1 (Slowly)

層 Low 内の activate ブロック内で，層 Low が活性化された時刻を計り，層 Low 内の press メソッド内で，press メソッドが呼び出された際の時刻と比較することで，層 Low が活性化されてから press メソッドが呼びだされるまでの時間を出力している．層 Low の状態へと直ちに（ここでは Low が活性化してから 2431 ミリ秒後に）再び press を呼び出すと，Low が非活性化され Bright が活性化される（図 5）．再び press を呼び出すと，Bright が非活性化され Off が活性化される．

2.2 デスクランプ Ver.2

デスクランプ Ver.1 のシステムを拡張したシステムであるデスクランプ Ver.2 を実装する．デスクランプ Ver.2 では，デスクランプ Ver.1 のシステムに対して新たに Low の状態から一定時間経過後にランプが Off の状態になる機能を追加する．デスクランプ Ver.2 の時間オートマトンを**図 7** に示す．

新たなロケーション FadeOut とイベント change が追加されている．FadeOut には Invariant が付与されている．Invariant はロケーションに対する注釈であり，そのロケーションに滞在可能な条件を指定することができる．例えば，ここでは FadeOut には Invariant として t<=55 が与えられている．FadeOut に滞在していられる条件はクロック変数 t が 55 単位時間以内の間であることを意味する．この Invariant と FadeOut から Off への辺に付く Guard t>=45 と組み合わせることにより，FadeOut に遷

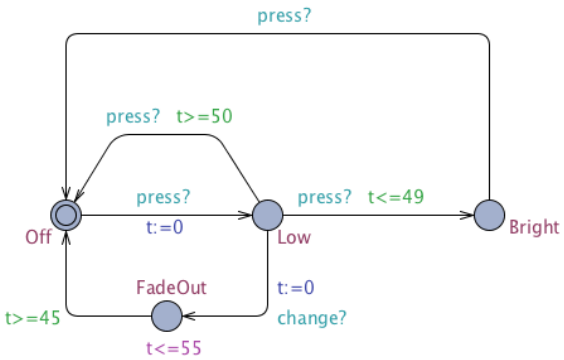

図 7 デスクランプ Ver.2 の時間オートマトン
Fig. 7 Timed automaton of DeskLamp Ver.2

```
1  process Lamp2 extends Lamp1{
2     event change: * Lamp.change();
3     layer Low{
4        transition t1: change? then FadeOut;
5     }
6     layer FadeOut{
7        within t2 [45, 55]{
8           then Off;
9        }
10    }
11 }
```

図 8 デスクランプ Ver.2 のプロセス記述
Fig. 8 Process description of DeskLamp Ver.2

移してから 45 単位時間以上 55 単位時間以内に Off へと遷移することを意味する．

図 7 はプロセス記述を用いると**図 8** のように表せる．ProcneJ では拡張部分のみを記述することが可能である．よって，Lamp1 プロセスから拡張したい部分のみを Lamp2 プロセスに記述する．extends Lamp1 と記述することにより，オブジェクト指向の継承と同様の意味を持たせることができる．

2 行目では新たなイベント change を宣言している．3~5 行目ではデスクランプ Ver.1 で作成した Lamp1 プロセスで定義された層 Low への追加の遷移規則が加えられている．6~10 行目では新たな層 Fadeout が定義されている．7~9 行目では within ブロックを用いて遷移規則を定義してる．within はこの層を活性化していられる時間を指定できる遷移規則である．within は FadeOut へ遷移してから 45 単位時間から 55 単位時間の間に Off へ遷移することを意味している．

デスクランプ Ver.2 のクラス記述 (**図 9**) を実行例 (**図 10**) と共に見ていく．press メソッドの呼び出しに対する振る舞いはデスクランプ Ver.1 と同様である．デスクランプ Ver.1 と共通する部分は省略してある．層 FadeOut 内の activate ブロックと deactivate ブロック内では現在時刻を調べ，層 FadeOut が活性化してから非活性化する

```
1  public class Lamp{
2     ...
3     layer Off{...}
4     layer Low{
5        ...
6        public void change(){
7           print("Change in Low");
8        }
9     }
10    layer Bright{...}
11    layer FadeOut{
12       activate{
13          activateTime=currentTime();
14          print("Activate FadeOut");
15       }
16       deactivate{
17          print("Deactivate FadeOut");
18          print((currentTime()-activateTime)+" ms");
19       }
20    }
21 }
```

図 9 デスクランプ Ver.2 のクラス記述
Fig. 9 Class description of DeskLamp Ver.2

```
Activate Off
Press 'p' or 'c': p
Press in Off
Base Press
Deactivate Off
Activate Low
Press 'p' or 'c': c
Change in Low
Deactivate Low
Activate FadeOut
Press 'p' or 'c': Deactivate FadeOut
4460 ms
Activate Off
```

図 10 デスクランプ Ver.2 の出力結果
Fig. 10 Result of DeskLamp Ver.2

までの時間を計っている．Low が活性化されている状態で，change メソッドを呼び出す（c を入力する）と，状態が Low から FadeOut に遷移する．出力結果を見ると，新たに p や c の入力を待っている間に，45~55 単位時間が経過したので，自動的に FadeOut から Off へと遷移していることがわかる．

2.3 デスクランプ Ver.3

デスクランプ Ver.2 にさらに拡張を加えたデスクランプ Ver.3 を実装する．デスクランプ Ver.2 では一定時間経つと Off へと遷移していたが，デスクランプ Ver.3 では光量を 10 単位時間毎に一定の割合で減らしていき，最終的に電気を消す（Off へと遷移する）ように改良する．デスクランプ Ver.3 を表す時間オートマトンを**図 11** に示す．

デスクランプ Ver.3 では，新たに光量を表す変数

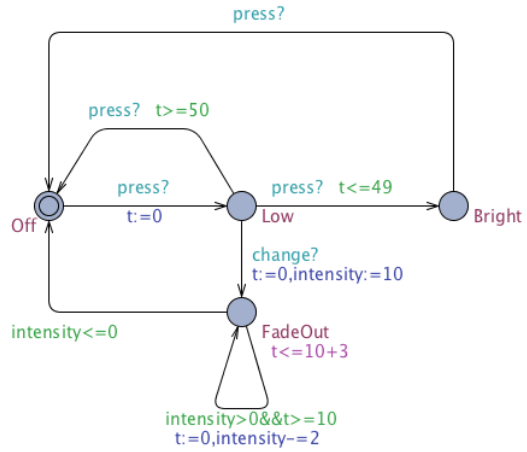

図 11 デスクランプ 3 の時間オートマトン
Fig. 11 Timed automaton of DeskLamp Ver.3

intensity と FadeOut に自身へと遷移する辺を追加した．この辺と FadeOut に付加された Invariant により，intensity の値が 0 以下になるまで 10 単位時間以上 13 単位時間以下の間隔で intensity の値を 2 ずつ減らす周期的な動作を行う．intensity の値が 0 以下になった場合には Off へと遷移するようになっている．

図 11 の時間オートマトンをプロセス記述を用いて表すと**図 12** のようになる．デスクランプ Ver.2 で定義した遷移の定義の変更や削除をするために，デスクランプ Ver.2 の層 Low の遷移規則に付けた名前 t1 と層 FadeOut で定義した遷移規則に付けた名前 t2 を使用して変更，削除を行う．

4 行目では Lamp2 プロセスで定義した遷移規則 t1 に対して新たに with{intensity=10;}を追加している．これは時間オートマトンの Update に対応する記述である．7 行目では，デスクランプ Ver.2 の層 FadeOut 内で定義した遷移規則 t2 を取り除いている．8~9 行目は，指定された単位時間毎に繰り返し行う遷移規則を periodic を用いて定義している．periodic の第一パラメータ 10 は周期の間隔を意味する．第二パラメータ 3 は，周期の jitter を指

```
1  process Lamp3 extends Lamp2{
2     declared int intensity=0;
3     layer Low{
4        modify t1: with{intensity=10;};
5     }
6     layer FadeOut{
7        remove t2;
8        periodic(10, 3): when(intensity>0)
9                         with{intensity-=2;};
10       transition: when(intensity<=0) then Off;
11    }
12 }
```

図 12 デスクランプ Ver.3 のプロセス記述
Fig. 12 Process description of DeskLamp Ver.3

```
1  public class Lamp{
2     process declared int intensity;
3     ...
4     public void showIntensity(){
5        print("Intensity␣:␣" + get_intensity());
6     }
7     layer Off{...}
8     layer Low{...}
9     layer Bright{...}
10    layer FadeOut{
11       activate{
12          activateTime=currentTime();
13          print("Activate␣FadeOut");
14          showIntensity();
15       }
16       deactivate{
17          print("Deactivate␣FadeOut");
18          print((currentTime()-activateTime)+"␣ms");
19       }
20    }
21 }
```

図 13 デスクランプ Ver.3 のクラス記述
Fig. 13 Class description of DeskLamp Ver.3

定している．when(intensity>0) は時間オートマトンの Guard に相当する．この遷移規則は，intensity>0 である限り，intensity-=2; を 10~13 単位時間毎に行うことを示している．10 行目では，intensity<=0 になった場合に Off へと遷移することが定義されている．

デスクランプ 3 のクラス記述を**図 13** に示す．2 行目ではプロセスで宣言された変数である intensity を宣言している．この変数はクラス記述では読み取り専用の変数である．クラス記述から読み取る際には get_intensity() のような getter を使用する．4~6 行目では新たに showIntensity メソッドを定義している．showIntensity メソッドは，先ほどの getter を使用して intensity の値を出力する．FadeOut 層の activate ブロックでは新たに，活性化された際に，showIntensity を呼び出し，intensity の値を出力するようにしている．

また，実行結果を**図 14** に示す．FadeOut 層が活性化されるまでの出力結果はデスクランプ Ver.2（図 10）のものと同じであるので省略している．Low の状態にいる時に change メソッドを呼び出す（c を入力する）と FadeOut に遷移する．新たに p や c の入力を待っている間に，再び FadeOut が非活性化と活性化を繰り返し，10 単位時間毎に intensity が 2 ずつ減少していることが出力結果に示されている．

3. 実装

3.1 クラス記述の変換

クラス記述のトランスレータは JastAddJ[5] を用いて実装され，クラス記述を Java へと変換する．

```
.
.
.
Activate FadeOut
Intensity : 10
Press 'p' or 'c': Deactivate FadeOut
912 ms
Activate FadeOut
Intensity : 8
Deactivate FadeOut
999 ms
Activate FadeOut
Intensity : 6
Deactivate FadeOut
1001 ms
Activate FadeOut
Intensity : 4
Deactivate FadeOut
1000 ms
Activate FadeOut
Intensity : 2
Deactivate FadeOut
1000 ms
Activate FadeOut
Intensity : 0
Deactivate FadeOut
100 ms
Activate Off
```

図 14 デスクランプ 3 の出力結果
Fig. 14 Result of DeskLamp Ver.3

各層は内部クラスへと変換される．層内で定義された activate ブロックと deacitvate ブロックはそれぞれメソッドへと変換され，各層が活性化/非活性化された時に呼び出される．また，層内で定義された部分メソッドは内部クラスのメンバメソッドとなる．部分メソッド内で呼び出される proceed() メソッドは，トランスレータによって同時に活性化されている層の同名メソッドを呼び出す処理へと変換される．さらに，現在活性化されている層を管理するためのフィールドや，メソッドが呼びだされた際に現在活性化されている層内で定義された部分メソッドの呼び出しに関する実装がトランスレータによって層を持つクラス内に追加される．これらの COP に関する実装については EventCJ の実装を参考にしている．

トランスレータは，層を持つクラスに COP に関する部分以外に内部クロックを扱う実装を加える．**図 15** は，トランスレータによって Lamp クラスに追加される内部クロックに関する部分の実装である．トランスレータによって long 型の変数__clock_が層を持つクラスのメンバ変数として宣言される．クラスのインスタンス化と同時に，__clock_を初期化する．ScheduledExecutorService は，指定した時間後または定期的にコマンドを実行するようにスケジュールできる．scheduleAtFixedRate は，指定した初期遅延の後に有効になり，その後は指定した期間毎に実行される定期的なアクションを作成し実行する．ここでは，

```
1  public class Lamp implements Contextual, Runnable{
2     ...
3     public Lamp(){
4        ...
5        __clock_=0;
6        ScheduledExecutorService ex;
7        ex=Executors.newSingleThreadScheduledExecutor
             ();
8        ex.scheduleAtFixedRate(this, 0, 100,
9                               TimeUnit.MILLISECONDS
10                             );
11    }
12    private long __clock_;
13    public long getClock(){return __clock_;}
14    public void run(){__clock_++;}
15 }
```

図 15 Lamp クラス内の内部クロックに関する実装
Fig. 15 Implementation of the internal clock in Lamp class

100 ミリ秒毎に__clock_をインクリメントする処理を行なっている．これにより，__clock_が時間の経過とともに増加していく．プロセス記述から生成される AspectJ のコードで__clock_の値を参照し，時間制約を表現している．

3.2 プロセス記述の変換

プロセス記述は実行のための AspectJ のコードと検証のための時間オートマトンへと変換される．プロセス記述のトランスレータは Eclipse の DSL 開発フレームワークである Xtext[6] を使用して実装を行った．ProcneJ のトランスレータでは，AspectJ や時間オートマトンへと変換する前に，複数のプロセスの合成と継承元のプロセスの削除や変更を行う．その後，before，after，periodic，within といった時間制約に関する規則を時間変数を用いた表現へと変換する．その後，各形式への変換を行う．

例えば，図 12 の Lamp3 プロセスは Lamp1，Lamp2 プロセスの差分のみを記述している．実際には Lamp3 プロセスはトランスレータによって**図 16** のように変換される．

プロセス記述における before，after，periodic，within といった各時間制約に関する記述は，直接プロセス記述の Invariant や Guard や transition を使用した遷移規則へと変換されるマクロの役割を果たしている．例えば，Lamp3 プロセスの before は，この before を持つ遷移規則に対する Guard に指定した時間制約を付け加える．

```
transition: before(49) press? then Bright;
```

の遷移規則の場合，

```
transition: when(__pcjLamp3BAAClock__<=49)
            press? then Bright;
```

のように変換される．ここで__pcjLamp3BAAClock__

```
process Lamp3{
   event press: * Lamp.press();
   event change: * Lamp.change();
   init layer Off{
      transition: press? then Low;
   }
   layer Low{
      transition: after(50) press? then Off;
      transition: before(49) press? then Bright;
      transition t1: change? then FadeOut
                     with{intensity=10;};
   }
   layer Bright{
      transition: press? then Off;
      transition: change? then FadeOut
                  with{intensity=20;};
   }
   layer FadeOut{
      periodic(10, 3): when(intensity>0)
                       with{intensity-=2;};
      transition: when(intensity<=0) then Off;
   }
}
```

図 16 合成後の Lamp3 プロセス
Fig. 16 The generated process description for Lamp3

```
after(Lamp l): execution(* Lamp.press()) &&
               target(l){
   InstanceLayerManager lm=lamp.getLayerManager();
   if(lm.isActive("Low")&&
      ((l.getClock()-__pcjLamp3BAAClock__)>=50)){
      lm.deactivate("Low").activate(
         l.createLayerForName("Off"));
   }
   ...
}
```

図 17 press イベントに関するポイントカットとアドバイス
Fig. 17 The pointcut/advice for press event

プロセス	時間オートマトン
InitLayer	初期状態のロケーション
Layer	ロケーション
Transition	辺
Invariant	Invariant
Sync	Synchronasation
Guard	Guard
Update	Update

表 1 プロセスと時間オートマトンの変換表
Table 1 The translation table from processes to timed automata

は，各プロセスにトランスレータによって宣言されたクロック変数である．この層へと遷移する全ての遷移規則に__pcjLamp3BAAClock__に 0 を代入する振る舞いがトランスレータによって付け加えられるので，この層へと遷移してから 49 単位時間以内に press イベントを受信すれば Bright へと遷移する条件となる．

こうして得られたプロセス記述（**図 A·1**）がさらにトランスレータによって AspectJ と時間オートマトンへと変換される．まず，AspectJ への変換について述べる．

プロセスはアスペクトへと変換され，プロセス内の遷移規則の各要素はポイントカットやアドバイスへと変換される．クロック変数はアスペクト内では long 型のローカル変数として定義される．イベントは，メソッドの実行に基づくプリミティブポイントカット execution として定義される．

図 17 は，press イベントを受信した際に起こるポイントカットとアドバイスの一部である．after アドバイスにより，ジョインポイントの実行後に遷移規則に沿った振る舞いを行う．さらに target ポイントカットによりジョインポイント実行時の処理対象を定め，処理対象となるインスタンスのみの層活性化制御を行う．インスタンスの活性化されている層を管理している LayerManager を取得し，層の活性化や非活性化の操作する．クロック変数への代入は，代入する値をクラス記述から Java への変換時に生成されたフィールド__clock_の値から減算した値を代入する．時間制約のチェック時には，その時点での__clock_の値とクロック変数の値の差により，時間経過とともに増加した場合のクロック変数の値を得ることができる．

次に時間オートマトンへの変換について述べる．プロセス記述（図 A·3）と時間オートマトンの対応表を**表 1** に示す．この表に沿ってプロセス記述から時間オートマトンに変換される．また，プロセス記述内で宣言された変数やイベントも時間オートマトンのグローバル変数やローカル変数，パラメータへと変換される．

4. 例題

シンプルなトースターの例 [7] を使用し，ProcneJ を用いた場合と用いない場合とで比較し，ProcneJ の有用性を示す．

4.1 シンプルなトースター

一回の調理で食パンが一枚のみセットでき，調理時間を設定して焼き加減を調節できるようなトースターのシステムを考える．システムは以下の様な特性を持っている．

- 調理時間はタイマセットボタンである二つのボタンを押すことにより 30 秒単位で調理時間を設定できる．
  - インクリメントボタン（以降 I ボタン）を押すことによりタイマの秒数を 30 秒増加する．
  - デクリメントボタン（以降 D ボタン）を押すことによりタイマの秒数を 30 秒減少させる．
- 調理開始ボタン（以降 S ボタン）は食パンと調理時間がセットされていれば調理を始める．

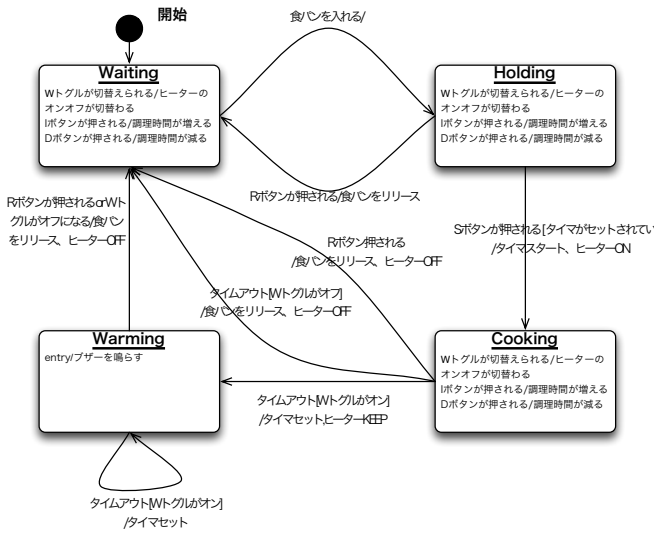

図 18 トースターのステートチャート
Fig. 18 The state chart of Toaster

```
1  public class Warming extends ToasterState{
2     private static ToasterState _instance=new
          Warming();
3     private Warming(){}
4     public static ToasterState getInstance(){
5       return _instance;
6     }
7     public void release(Toaster toaster){
8       toaster.toasterAPI.heaterOff();
9       toaster.toasterAPI.upBread();
10      toaster.stopBuzzTimer();
11      toaster.changeState(Waiting.getInstance());
12    }
13    public void toggle(Toaster toaster){
14      toaster.toasterAPI.togglePush();
15      toaster.toasterAPI.heaterOff();
16      toaster.toasterAPI.upBread();
17      toaster.stopBuzzTimer();
18      toaster.changeWarmState();
19      toaster.changeState(Waiting.getInstance());
20    }
21 }
```

図 19 Warming クラス
Fig. 19 Warming class

- リリースボタン（以降 R ボタン）は食パンが入っている状態で押すと食パンがリリースされる．
- 調理後に保温するかどうかは保温トグル（以降 W トグル）の状態によって決まる．
  - W トグルを ON に設定すると，食パンの調理が終わった後も一定温度で保温される．
  - W トグルが OFF に設定されている場合には，調理が終わった後すぐに食パンがリリースされる．
  - 保温中は一定時間毎にブザーが鳴り，保温中に W トグルが ON から OFF にされた場合はヒーターが切られ食パンがリリースされる．

実装するトースターシステムのステートチャートを**図 18** に表す．各状態での動作を説明する．

Waiting 状態は初期状態であり，食パンが入っていない状態を指す．この状態では W トグルの切り替え，調理時間の設定を行うことができる．食パンがセットされると *Holding* 状態へと遷移する．

Holding 状態は食パンがセットされた調理前の状態である．*Waiting* 状態と同様に W トグルの操作と調理時間の設定ができる．また，R ボタンが押された場合には食パンがリリースされ，*Waiting* 状態に戻る．調理時間が 0 より大きい状態で S ボタンが押されると，調理を開始し *Cooking* 状態へと遷移する．

Cooking 状態は食パンが調理されている最中の状態を表す．*Cooking* 状態でも，調理時間の調整や W トグルの切り替えを行うことができる．R ボタンが押されると調理を中止し食パンをリリースして *Waiting* 状態に戻る．指定した時間が経過したり，D ボタンが押されたことにより調理の残り時間が 0 になると，W トグルの状態によって動作と状態の遷移先が変わる．W トグルが OFF に設定されている場合には，食パンをリリースし *Waiting* 状態に戻る．W トグルが ON に設定されている場合には，タイマをブザーを鳴らす間隔にセットして *Warming* 状態に遷移する．

Warming 状態は食パンの調理が終わり，保温状態にあることを指す．この状態に遷移したら，まずブザーを鳴らしてヒーターを保温状態にする．*Warming* 状態では W トグルが ON である限り一定時間毎に再び *Warming* 状態に再び戻ってくる．R ボタンが押されるか，W トグルが OFF にされると食パンをリリースして *Waiting* 状態に遷移する．

4.2 ProcneJ を使用しない実装

図 18 を元にトースターのシステムを Java を用いて実装する．トースターの各状態をクラスとして表現して各状態に依存する振る舞いをモジュール化した．

Warming 状態を表す `Warming` クラス（**図 19**）について説明する．`Warming` クラスでは *Warming* 状態の振る舞いが定義される．`ToasterState` クラスはトースターの各状態を表すクラスのスーパークラスである．`toggle` メソッド内では *Warming* 状態の時に W トグルの切り替えが行われた場合に保温状態が OFF になるので，`Warming` クラス内の `toggle` メソッドにはヒーターを止めて食パンをリリース振る舞いと，ブザーを止めて *Waiting* 状態に戻る振る舞いが定義されている．`release` メソッドにも同様にヒーターの停止と食パンをリリースする振る舞いに加えて，ブザーを止める振る舞いが定義されている．

`startBuzztimer` メソッドは周期的にブザーを鳴らす振る舞いが定義され，*Cooking* 状態から *Warming* 状

```
1  public class Toaster{
2     ...
3     private int buzzTime=100;
4     ScheduledExecutorService ex;
5     ...
6     protected void startBuzzTimer(){
7        ex=Executors.newSingleThreadScheduledExecutor
             ();
8        Runnable pinger=new Runnable(){
9           public void run(){toasterAPI.buzzSound();}
10       };
11       ex.scheduleAtFixedRate(pinger, 0,
12                              buzzTime*100,
13                              TimeUnit.MILLISECONDS
14                             );
15    }
16    protected void stopBuzzTimer(){ex.shutdown();}
17 }
```

図 20 Toaster クラス
Fig. 20 Toaster class

態へと遷移する際に呼びだされる．時間制約に関する実装は Toaster クラス（**図 20**）で行なっている．startBuzzTimer メソッドでは 100 ミリ秒毎にブザーを鳴らす処理を行なっている．stopBuzzTimer メソッドを呼び出すことにより，ブザーを停止させることができる．

4.3 ProcneJ を使用した実装

トースターのクラス記述を**図 21** に示す．クラス記述ではトースターの状態毎に層を使って実装し，それぞれの層でトースターが同名の状態にある時の振る舞いを定義している．各 activate ブロックでは，活性化された時の処理が定義されている．activate ブロックや deactivate ブロックを使用することにより，各状態に遷移するとともに行いたい振る舞いをモジュール化して定義することができる．Warming 層の release メソッドでは，共通部分を proceed メソッドで呼び出しているのに加え，ヒーターをオフにする処理を加えている．

トースターのプロセス記述を**図 22** に示す．*Waiting*，*Holding*，*Cooking*，*Warming* の各状態を表す 4 つの層があり，それぞれの時間制約と層遷移規則がこのプロセス記述に定義されている．Warming 層では buzzTime で指定した単位時間毎に，ブザーを鳴らすために periodic を使用して遷移規則を定義している．

4.4 検証

トースターシステムのプロセス記述から生成された時間オートマトンを**図 23** に示す．

この時間オートマトンに対して UPPAAL 上で検証を行うためにプロセス記述のテストプロセスを用いて検証のための時間オートマトンを作成する．検証したいユーザの動

```
1  public class Toaster{
2     protected ToasterAPI toasterAPI;
3     process declared int cookTime;
4     ...
5     public void release(){toasterAPI.upBread();}
6     public void toggle(){toasterAPI.togglePush();}
7     layer Waiting{...}
8     layer Holding{...}
9     layer Cooking{...}
10    layer Warming{
11       activate{
12          toasterAPI.heaterKeep();
13          toasterAPI.buzzSound();
14       }
15       deactivate{}
16       public void toggle(){
17          toasterAPI.heaterOff();
18          proceed();
19       }
20       public void release(){
21          toasterAPI.heaterOff();
22          proceed();
23       }
24    }
25 }
```

図 21 トースターシステムのクラス記述
Fig. 21 Class description of Toaster

```
1  process Toaster{
2     declared int cookTime = 0;
3     bool warmState = false;
4     int buzzTime = 150;
5     ...
6     event release: * Toaster.release();
7     event toggle: * Toaster.toggle();
8
9     init layer Waiting{...}
10    layer Holding{...}
11    layer Cooking{...}
12    layer Warming{
13       periodic(buzzTime, 10):;
14       transition: release? then Waiting with{
              cookTime=0;};
15       transition: toggle? then Waiting with{
              warmState=!warmState;};
16    }
17 }
```

図 22 トースターシステムのプロセス記述
Fig. 22 Process description of Toaster

作を時間オートマトン（**図 24**）を用いて表現する．このユーザはまず食パンをトースターに挿入し，I ボタンを二回押す．I ボタンを二回押したので約 600 単位時間調理することとなる．その後，S ボタンを押して調理を開始させる．300 単位時間経過したあたりで W トグルを押し ON の状態にする．調理が終了するとトースターは保温状態になる．調理が終了し一定時間経過した後，ユーザが W トグルを

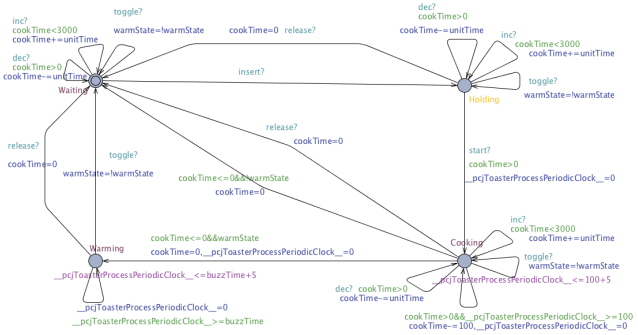

図 23 トースターの時間オートマトン
Fig. 23 Toaster Timed Automaton

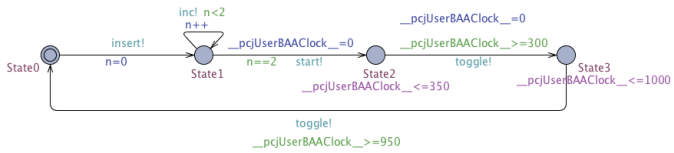

図 24 ユーザの時間オートマトン
Fig. 24 User Timed Automaton

再び押すことで食パンが保温が終了し食パンが出てくる．またトースターは保温状態から待機状態へと遷移する．

この時間オートマトンをプロセス記述のテストプロセスを用いて表すと**図 25** になる．テストプロセスはプロセスとほぼ同じ意味合いを持つ．異なる点はテストプロセスは時間オートマトンにしか変換されないことと `target` の存在である．`target` は，どのプロセスを対象としたテスト用の時間オートマトンかを指定することができ，対象としたプロセスの内で宣言されたイベントや定義を使用することができる．ここでは `target Toaster` として `Toaster` プロセスを対象としているので，`Toaster` プロセス内で定義されているイベントを宣言なしに使用している．

UPPAAL では簡易版の TCTL を用いてシステムの特性の検証をすることが可能である．今回作成した時間オートマトンに**表 2** のような特性を検証した．

今回作成したトースターシステムの開発で ProcneJ の利用における利点は二点挙げられる．まず，プロセス記述には層遷移規則，ここではトースターの状態の変化の規則について実際に動作する部分が，ProcneJ を使用した実装では完全に分離できている．また，時間制約に関する部分の記述もプロセス記述側にモジュール化することができている．次に，検証が可能という点では ProcneJ を使用しない場合では，テストを繰り返して時間制約を満たしているかのチェックを行うことになるが，ProcneJ では，実装した時間制約に関する実装に問題がないかの検査を自動的に生成される時間オートマトンによってモデル検査することが

```
1  test User target Toaster{
2     int n;
3     init layer State0{
4        transition: insert! then State1 with{n=0;};
5     }
6     layer State1{
7        transition: inc! when(n<2) with{n++;};
8        transition: start! when(n==2) then State2;
9     }
10    layer State2{
11       within[300, 350]{
12          toggle! then State3;
13       }
14    }
15    layer State3{
16       within[950, 1000]{
17          toggle! then State0;
18       }
19    }
20 }
```

図 25 ユーザのテストプロセス
Fig. 25 Test process of User

`A[] not deadlock`	Satisfied
`A[] user.State0 imply toaster.Waiting`	Satisfied
`A[] user.State1 imply toaster.Holding`	Satisfied
`A[] user.State2 imply toaster.Cooking`	Satisfied
`A[] user.State2 imply toaster.warmState`	Not Sateisfied
`A[] user.State3 imply toaster.warmState`	Satisfied

表 2 特性と検証結果
Table 2 Properties and verification results

可能である．ProcneJ を実時間システムの開発で使用することで，実際の動作に関するコードの記述と，状態遷移や時間制約に関する部分のコードの記述が混在しないという点と時間制約やシステムの状態の遷移について検証できるという点において有用性があると評価できる．

5. 関連研究

5.1 モデルベース開発

モデルからコードを生成する技術をモデルベース開発と呼ぶ．モデルベース開発の手法を実時間システムの性質の検証に応用し，検証されたモデルから実際に実行可能なコードを生成する手法も研究されている．

Pour らの研究 [8] では，モデルを記述するために時間オートマトンを使用している．時間オートマトンで記述されたモデルは，モデル検査器 UPPAAL で機械的に検証し，Real-Time Specification for Java (RTSJ)[9] で書かれた実時間プログラムにマッピングされる．同様に UPPAAL を用いて時間オートマトンを作成し，そこから標準の Java のコードを生成する Senthooran らの研究 [10] が挙げられる．

これらの研究では，UPPAAL のグラフィカルなユーザイ

ンタフェースを使用してモデルの作成，検証を行い，検証したモデルからコードが自動的に生成されるという利点がある．一方で，UPPAAL の時間オートマトンはモジュール化の手法などがなく，システムの規模の巨大化，複雑化とともに時間オートマトンの作成は困難になる．また，システムの時間オートマトンのモデルから実行可能なコードを生成 しているが，生成されるのは時間オートマトンで表現可能な範囲のモデルから得られるコードであって，実行可能な完全なコードではない．実際には生成されたコード中に実際の実行に必要なコードを記述していかなければならないという問題点も存在する．モデルと生成されたコードのギャップは，デバッグやモデルから生成されたコードとそうでないコードとの混用を困難にする．

5.2 文脈指向プログラミング言語

ContextJ[11] は Java に文脈指向プログラミングの考えを導入した文脈指向プログラミング言語である．層と部分メソッドを使用して文脈に依存する振る舞いをモジュール化することができる．加えて，`with/without` ブロックや `activate/deactivate` メソッドを使用して層活性化制御を行なっている．プログラマの任意の場所で層の活性化ができるという利点はあるが，層の活性化に関する記述がコード内で散在してしまうという問題点もある．

EventCJ[2] もまた ContextJ と同様に Java に対して文脈指向プログラミングの概念を導入した文脈指向プログラミング言語である．ContextJ と異なる点は，層活性化制御の方法であり，プログラムの実行ポイントをイベントと見なし，イベントの発生時に層の活性化を行う．プログラムの実行ポイントと層の遷移規則はイベント宣言と層遷移規則を用いて宣言的に指定することが可能である．この方法ではブロック文やメソッドを使用して活性化制御を行う方法に比べ，層の活性化制御に関するコードの分離が可能となり，文脈に依存する振る舞いを記述と層の活性化に関する記述が混在しないという利点がある．一方，ContextJ のブロックやメソッド呼び出しの手法では層の活性化が確実に行われることが保証されていたが，層遷移規則の記述の仕方によっては期待する活性化が得られない場合もある．そこで，EventCJ では層遷移規則から SPIN モデル検査器用のコードを生成し，層遷移の特性を検証できるようにしている．

Subjective-C[12] は Objective-C 上に文脈指向プログラミングの概念を導入した文脈指向プログラミング言語である．Subjective-C は層と部分メソッドを定義するのではなく，メソッドに対してアノテーションをつけることにより依存する文脈を明示させる．また，メソッド呼び出しによる活性化制御を採用している．Subjective-C 独自の特徴として文脈同士の関係の宣言が可能な点である．層同士の弱い包含，強い包含，排斥関係，要求関係の 4 種類の関係の記述を可能にし，層の相互依存を表せるようになっている．

6. 今後の課題

本研究で行った実時間システムの検証は要求や設計に関する検証のみの時間的な要件の性質が満たされているかの検証であり，実際の実装がその仕様を満たしているのかの検証は行なっていない．利用可能な資源の上で要求されるタスクのスケジューリングの可能性を考慮した記述の導入などが必要となる．

本研究で提示した例題ではイベントは全て，ユーザボタンがボタンを押したなどの外部から起きたイベントに対する受信のみであり，現在の ProcneJ では，プロセスからのイベントの送信は実現不可能であり，イベントはメソッドが外部から呼び出された場合にしか対応出来ない．プロセスから別のプロセスへのイベントの送信を行い，プロセス同士で同期することを可能にすることで，さらに複雑な仕様の記述をサポートすることが可能となる．

また，プロセス記述はプロセスの継承と差分の記述をサポートしているが，クラス記述は継承や差分の記述をサポートしていない．プロセス記述の拡張に伴い，クラス記述も拡張される．層によってモジュール化が成されているので新たな層や振る舞いの追加や削除は行い易いが，差分として記述することが可能になれば，プロセス記述とクラス記述の対応が取れ，コードの可読性の良くなり，さらに拡張しやすくなると考えられる．

さらに，トースターの例では，調理時間の増減を行った際の状態遷移や保温トグルを操作した際の状態遷移はそれぞれ自身の層への遷移をそれぞれの層へと記述する必要があった．しかし，複数の層を抽象化した複合層の記述を導入し，重複する遷移規則の記述を複合層で行うことを可能にすることは，各層で共通する層遷移規則の記述の冗長性の解消に繋がる．

7. おわりに

本発表の目的は新しい時間制約の記述方式とそのモジュール化手法，およびそれに基づく言語の提案であり，本論文では提案手法に基づいて設計した ProcneJ とその実装について説明した．

提案手法を用いることにより，時間制約の記述のモジュール化を可能にし，さらに記述したコードから検証可能なモデルを生成し，時間制約や状態遷移などの仕様の特性の検証が可能となった．また，モデルベース開発を応用した先行研究では，形式的なモデルから実行可能なコードの生成が行われているが，これにはシステムが複雑になるとモデルとコードのギャップが大きくなる問題点が存在した．しかし提案手法により，よりモデルに近い形でコードを記述することが可能になり，モデルとコードのギャップを縮めることが可能になった．

今後，ProcneJ を使用して様々な実時間システムの実装を行い，課題の解決策を考え ProcneJ の改良を行なっていく．

参考文献

[1] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, Vol. 7, No. 3, pp. 125–151, 2008.

[2] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: A context-oriented programming language with declarative event-based context transition. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD '11)*, pp. 253–264. ACM, 2011.

[3] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Journal on Software Tools for Technology Transfer*, Vol. 1, No. 1–2, pp. 134–152, October 1997.

[4] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pp. 327–353, London, UK, UK, 2001. Springer-Verlag.

[5] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA 2007)*, pp. 1–18, 2007.

[6] Xtext. http://www.eclipse.org/Xtext/.

[7] Rob Williams. リアルタイム組込みシステム基礎講座. 翔泳社, 2006.

[8] Niusha Hakimipour, Paul Strooper, and Andy Wellings. TART: Timed-automata to real-time Java tool. In *8th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2010)*, pp. 299–309. IEEE, 2010.

[9] Greg Bollella, James Gosling, Ben Brosgol, Peter Dibble, Steve Furr, David Hardin, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[10] Ilankaikone Senthooran and Takuo Watanabe. A model-based approach to constructing safe soft real-time programs for non-real-time environments. In *13th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2012)*, pp. pp.269–274. ACIS, 2012.

[11] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. *Computer Software*, Vol. 28, No. 1, pp. 272–292, 2011.

[12] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-c: Bringing context to mobile platform programming. In *Software Language Engineering (SLE 2010)*, Vol. 6563 of *Lecture Notes in Computer Science*, pp. 246–265. Springer-Verlag, 2010.

付　　録

付録として 3.2 節で得られた時間制約マクロを含むプロセスがトランスレータによって変換された後のプロセス記述と，クラス記述とプロセス記述の各構文を示す．構文の記述では，大文字から始まるイタリック体は文法要素，小文字のイタリック体はメタ変数，タイプライタ体は文法をあらわす．

A.1 時間制約マクロ変換後のプロセス記述

```
1  process Lamp3{
2     clock __pcjLamp3BAAClock__;
3     clock __pcjLamp3PeriodicClock__;
4     event press: * Lamp.press();
5     event change: * Lamp.change();
6     init layer Off{
7        transition: press? then Low
8                    with{__pcjLamp3BAAClock__=0;};
9     }
10    layer Low{
11       transition: when(__pcjLamp3BAAClock__>=50)
12                   press? then Off;
13       transition: when(__pcjLamp3BAAClock__<=49)
14                   press? then Bright;
15       transition t1: change? then FadeOut
16                      with{intensity=10;
17                      __pcjLamp3PeriodicClock__=0;};
18    }
19    layer Bright {
20       transition: press? then Off;
21       transition: change? then FadeOut
22                   with{intensity=20;
23                   __pcjLamp3PeriodicClock__=0;};
24    }
25    layer FadeOut {
26       invariant: __pcjLamp3PeriodicClock__<=10+3;
27       transition: when(intensity>0&&
28                   __pcjLamp3PeriodicClock__>=10)
29                   then FadeOut
30                   with{intensity-=2;
31                   __pcjLamp3PeriodicClock__=0;};
32       transition: when(intensity<=0) then Off;
33    }
34 }
```

図 A·1 時間制約マクロ変換後のプロセス記述

Fig. A·1 Process description after time constraint macro translation

A.2 各記述の構文

```
Class            ::= class id {
                         (DeclaredVar)*
                         (vardecl)*
                         (methoddecl)*
                         (Layer)*
                     }
DeclaredVar      ::= process declared vardecl;
Layer            ::= layer id {
                         (Activate)?
                         (Deactivate)?
                         (LayerMethodDecl)*
                     }
Activate         ::= activate { (exp)* }

Deactivate       ::= deactivate { (exp)* }

LayerMethodDecl  ::= (after | before)? methoddecl
```

図 A·2 クラス記述の構文

Fig. A·2 Syntax of class description

```
ProcessDescript  ::= (Decl)*
                     Process?
                     TestProcess?
Process          ::= process id extends id {
                         (Decl)*
                         (InitLayer)?
                         (Layer)*
                     }
TestProcess      ::= test id target id extends id {
                         (Decl)*
                         (InitLayer)?
                         (Layer)*
                     }

Decl             ::= Clock | Event | DeclaredVar | vardecl | functiondecl
Clock            ::= clock id ;
Event            ::= event id: exp ;
DeclaredVar      ::= declared vardecl ;

InitLayer        ::= init Layer
Layer            ::= layer id { (Invariant)? (Within)? (Transition | Periodic | Remove | Modifier)* }
Invariant        ::= invariant exp ;
Within           ::= within id [ exp , exp ] { ((Guard)? then id (Update)?)+ }

Transition       ::= transition id? : (TimeConst)? (Sync)? (Guard)? (Destination)? (Update)? ;
TimeConst        ::= After | Before | After & Before
Sync             ::= id! | id?
Guard            ::= when(exp)
Destination      ::= then id
Update           ::= with{(exp)*}
After            ::= after(exp)
Before           ::= before(exp)

Periodic         ::= periodic(exp , exp) id? : (TimeConst)? (Event)? (Guard)? (Update)? ;

Remove           ::= remove id ;

Modifier         ::= TransitionModifier | PeriodicModifier
TransitionModifier ::= modify id : (Event)? (Guard)? then id (Update)? ;
PeriodicModifer  ::= modify id (exp? , exp?) : (TimeConst)? (Event)? (Guard)? (Update)? ;
```

図 A·3 プロセス記述の構文

Fig. A·3 Syntax of process description