情報処理学会論文誌　プログラミング　Vol. 4　No. 3　26–41 (June 2011)

# Safe Ambients のための Java フレームワーク

岡田翔太[†1]　馬谷誠二[†1]　林　奉行[†1]
八杉昌宏[†1]　湯淺太一[†1]

アンビエント計算はプロセス代数の一種であり，並行プロセス間の協調動作や計算機間のコード移動は，アンビエントの移動動作として表現される．アンビエントの階層を用いることで，LAN，PC クラスタ，マルチコアプロセッサ，モバイルエージェントなどを統一的に表現可能な点が特徴である．本発表では，アンビエント計算の一種である Safe アンビエントに基づく，並列・分散計算のための Java フレームワークを提案する．本フレームワークを用いると，分散環境の構成要素をアンビエントとして統一的に記述し直接実行できる．アンビエントは通常の Java オブジェクトとして記述されるため，各アンビエントにインスタンス変数やメソッドを持たせることで，Java プログラマにとって分かりやすく簡潔なプログラムを書くことができる．フレームワークの実装においては，コード変換により JVM 間のアンビエントの移動を実現している．また単一 JVM 内においては，たとえば Java の入出力 API の完了待ちなどを考慮し，複数のプロセスは複数の Java スレッドにより並列実行されるのが望ましい．しかし，細粒度なプロセスを 1 対 1 に Java スレッドに対応づけるとプロセス生成のコストが大きくなる．本フレームワークは，Safe アンビエントの型情報を利用し，どのプロセス（またはアンビエント）を 1 つの Java スレッドに対応づけるかを適切に選択する．

# A Java Framework for Safe Ambients

Shota Okada,[†1] Seiji Umatani,[†1] Tomoyuki Hayashi,[†1]
Masahiro Yasugi[†1] and Taiichi Yuasa[†1]

The Ambient calculus is a kind of process calculi. Code migration among computers and synchronized movement among parallel processes are represented as movements of ambients. LANs, PC clusters, multicore processors, and mobile agents are uniformly represented as components of the hierarchy of ambients. In our presentation, we propose a Java framework for parallel and distributed computing, based on Safe Ambients. Each component of distributed environments is uniformly represented as an ambient and executed directly. Since each ambient is a normal Java object, it can contain instance variables and instance methods so that we can write programs in a practical manner and writing programs becomes easier. In the implementation of our framework, movements of ambients among JVMs are achieved by the code translation. In a single JVM, some events considered (e.g, waiting for finish of I/O APIs), multiple processes should be executed using several Java threads in parallel. However, creating a new Java thread for each fine-grain process at runtime is rather expensive. Therefore, our framework decides which process (or ambient) is mapped to a Java thread using type information of Safe Ambients.

## 1. はじめに

近年インターネットの発達により，分散環境で動くアプリケーションの開発がさかんに行われている．これらのアプリケーションにおいて，携帯性や移動性は重要な要素であるが，携帯性や移動性を兼ね備えた分散アプリケーションは，特に規模が大きくなるにつれ，開発が非常に困難になる．複数計算機間の複雑なやりとりを単に実現するだけでなく，プログラムの安全性，信頼性，正確性を求められるからである．そのようなアプリケーションの開発においては，形式的な分散計算モデルを用いて，分散環境内での動作を厳密に定義するのが望ましい．

アンビエント計算[4] はプロセス代数の一種であり，並行プロセス間の協調動作や計算機間のコード移動は，アンビエントの移動動作として表現される．アンビエントの基本動作には，in，out，open の 3 つがある．in は，アンビエントが別のアンビエントに入ることを表し，out は，アンビエントが自分の属しているアンビエントから出て行くことを表す．open は，アンビエントの境界の消失を表す．つまり，open されたアンビエントが含むプロセスを外のアンビエントから使えるようになる．

アンビエント計算を用いると，階層的な場所の概念を導入することができる．たとえば，LAN の中にサーバが 2 台，携帯電話が 1 台あるとき，それぞれをアンビエントととらえると，

$$LAN[server1[P] \mid server2[Q] \mid phone[R]]$$

と書ける．つまり，アンビエント計算は多様な分散計算環境をモデル化可能な分散計算モデルであるといえる．

†1 京都大学大学院情報学研究科
Graduate School of Informatics, Kyoto University

アンビエントは場所を表すことができるが，場所を表しているアンビエント自身も移動することができる．そのため，コードの移動を簡潔に書くことができる．たとえば，携帯電話をあるサーバに接続するような状況は，

$phone[\text{in } server1.R]$

と表現できる．

また，コードの安全性も実現できる．たとえば，

$phone[(\nu n)P]$

と書き，$P$ の中で名前 $n$ のアンビエントを作ると，そのアンビエントの名前 $n$ は外側から見ることができない．つまり，アンビエントの名前に匿名性を持たせることができる．

アンビエント計算には様々なバリエーションが存在するが，そのなかの 1 つに Safe Ambients（SA）[11] がある．SA では，上記の 3 つの動作に対して，それらと対になる動作（$\overline{\text{in}}$, $\overline{\text{out}}$, $\overline{\text{open}}$）が定義されている．ある動作を実行するとき，必ずそれと対の動作が必要となる点が特徴である．in，out，open 動作に対して，対の $\overline{\text{in}}$，$\overline{\text{out}}$，$\overline{\text{open}}$ 動作が存在するため，アンビエント間の協調動作をきめ細かに制御できる．たとえば，

$phone[\text{in } server1.P] \mid server1[\overline{\text{in}}\ server1.Q]$

と書くと，$P$，$Q$ は in，$\overline{\text{in}}$ 動作が終了してから実行される．アンビエントの動作を表現する in $a$ などは，ケーパビリティと呼ばれる．アンビエント間では，ケーパビリティを値として受け渡すことができるため，限られた能力のみを他のアンビエントに許可することが可能になる．たとえば，*server1* の名前を送信すると *server1* に対してすべての動作ができてしまうが，in *server1* というケーパビリティだけを送信することで，動作を制限することができる．

本研究では，SA に基づく並列・分散計算のための Java フレームワークを提案する．SA を，分散アプリケーションのモデルとその仕様を記述するためのツールとして用いるのではなく，実際の分散システムそのものを構築するための汎用プログラミング言語用ツールのベースモデルとして用いている点が本フレームワークの特徴である．先行研究として SA のプロセス式をそのまま解釈実行する処理系はいくつか存在するが[7)–9)]，それらはいずれも，汎用プログラミング言語が備えている現実のプログラムを書くのに必要な機能，あるいは汎用プログラミング言語で記述された外部コードと連携する機能を備えていないため，本フレームワークとは目的が異なっている．

本フレームワークは，SA の利点を活かしたフレームワーク設計となっており，SA が提案する型によるアンビエントの分類に基づいた適切な実装方式を選択している．さらに，実用的に記述できるようにするため，SA にいくつかの実用的な機能を追加している．たとえば，アンビエントは Java オブジェクトとして記述される．アンビエントを Java のオブジェクトとして扱うことができるため，各アンビエントにインスタンス変数やメソッドを持たせることができる．他にも，2 つの独立したアンビエント階層を 1 つのアンビエント階層に動的に接続する機能や，逆に，アンビエント階層の一部を動的に切り離す機能を追加している．

本稿の以降の構成は次のとおりである．2 章では SA についての説明を行い，3 章では仕様について述べる．4 章では具体的なプログラム例を用いて説明し，5 章では実装について述べる．6 章では関連研究を，7 章ではまとめと今後の課題を述べる．

## 2. Safe Ambients

Safe Ambients（SA）とはアンビエント計算の一種であり，in，out，open のプロセスの衝突を防ぐように設計されている．SA では in，out，open の動作が実行されるとき，それぞれ対になる動作（$\overline{\text{in}}$, $\overline{\text{out}}$, $\overline{\text{open}}$）が必要となる．対の動作が存在することにより，各命令が実行される順序を制御でき，同時実行可能な命令間の動作の不用意な干渉を防ぐことができる．

SA の構文は以下のとおりである．

$$M, N ::= x \mid n \mid in\ M \mid \overline{in}\ M \mid out\ M \mid \overline{out}\ M \mid open\ M \mid \overline{open}\ M$$

$$P, Q, R ::= 0 \mid P \mid Q \mid (\nu n)P \mid M.P \mid M[P] \mid \langle M \rangle \mid (x)P \mid X \mid rec\ X.P$$

以下，簡単に各構文要素について説明する．なお，$x$ は変数，$n$ はチャネル名，$X$ は再帰変数である．

- $P \mid Q$：プロセス $P$ とプロセス $Q$ を並列に実行する．
- $(\nu n)P$：$P$ 内部でのみ使用できる局所変数 $n$ を生成する．
- $M.P$：動作 $M$ を実行した後プロセス $P$ を実行する．
- $M[P]$：プロセス $P$ を内部に含む名前 $M$ のアンビエントを生成する．
- $\langle M \rangle$：送信プロセスを示す．$M$ というメッセージを送信する．
- $(x)P$：受信プロセスを示す．メッセージを受信してプロセス $P$ を実行する．

SA における主な動作は**図 1** のようになる．これらを組み合わせると，たとえば以下のような遷移が行われる．

$$a[b[\overline{\text{in}}\ b.\text{open}\ d.Q] \mid c[\overline{\text{out}}\ c.R \mid d[\text{out}\ c.\text{in}\ b.\overline{\text{open}}\ d.S]] \mid P]$$
$$\Rightarrow\ a[b[\overline{\text{in}}\ b.\text{open}\ d.Q] \mid c[R] \mid d[\text{in}\ b.\overline{\text{open}}\ d.S] \mid P]$$
$$\Rightarrow\ a[b[\text{open}\ d.Q \mid d[\overline{\text{open}}\ d.S]] \mid c[R] \mid P]$$

| | | |
|---|---|---|
| In | $m[\text{in } n.P_1 \mid P_2] \mid n[\overline{\text{in}}\ n.Q_1 \mid Q_2]$ | $\rightarrow n[m[P_1 \mid P_2] \mid Q_1 \mid Q_2]$ |
| Out | $m[n[\text{out } m.P_1 \mid P_2] \mid \overline{\text{out}}\ m.Q_1 \mid Q_2]$ | $\rightarrow n[P_1 \mid P_2] \mid m[Q_1 \mid Q_2]$ |
| Open | $\text{open } n.P \mid n[\overline{\text{open}}\ n.Q_1 \mid Q_2]$ | $\rightarrow P \mid Q_1 \mid Q_2$ |
| Message | $\langle M\rangle \mid (x)P$ | $\rightarrow P\{M/x\}$ |

**図 1**　遷移規則
Fig. 1　Reduction rules of actions in SA.

$\Rightarrow\ a[b[Q \mid S] \mid c[R] \mid P]$

## 3. 仕　　様

### 3.1 全体構成

本研究で提案するフレームワークは，2 章で述べた Safe Ambients に基づき構成されている．アンビエントは Ambient クラス（あるいは，そのサブクラス）のインスタンスとして表現される．アンビエントの名前は，Name クラスのインスタンスとして表現される．Ambient クラスと Name クラスとの関係を明確にするために，Ambient クラスと Name クラスは 1 対 1 で対応している．たとえば，ユーザが `A` という Ambient クラスのサブクラスを定義するならば，必ず `AName` という Name クラスが作られる．

本フレームワークではアンビエントを Ambient クラスとして表現しており，ユーザが新しいアンビエントを定義する際は Ambient クラスを継承して記述する．さらに，ユーザ定義のアンビエントクラスを定義する際，ユーザはインスタンス変数やインスタンスメソッドをその中に追加できる．これにより，インスタンス変数やインスタンスメソッドを通常どおり利用できることで既存の Java コードの多くを再利用可能となるという利点もある．インスタンス変数については，アンビエント内の局所通信や open 動作を駆使すれば原理的には同じことが実現可能だが，その場合コードはかなり煩雑にならざるをえず，Java プログラマにとって自然なインスタンス変数をそのまま使用できる方が望ましいと思われる．アンビエントクラス中のインスタンス変数の有用な使用例については次節で述べる．

アンビエントは IMAmbient（Immobile Ambient）と STAmbient（Single-Thread Ambient）の 2 種類に分類される．STAmbient は in，out，open などの動作が同時にたかだか 1 つしか実行可能にならないアンビエントのことを意味する．IMAmbient は in，out，$\overline{\text{open}}$ の動作を含まないアンビエントを意味する．それぞれ Ambient クラスのサブクラスであり，本フレームワークにおける実装方法もそれぞれで異なる（詳細は 5 章で述べる）．本来の SA [11] では，型推論システムにより，アンビエントが STAmbient であるか IMAmbient であるかを自動で判別するが，本フレームワークでは自動的に型推論を行う代わりに，Ambient クラスのサブクラスをそれぞれ用意している．ユーザ定義のアンビエントクラスは STAmbient か IMAmbient のいずれかを継承し，どちらに分類されるかを明示的に示さなければならない．

**表 1**　SA プロセスの API
Table 1　API of SA processes.

| Safe Ambients P | Java | 説明 |
|---|---|---|
| $P \mid Q$ | new Process(Ambient a){<br>public void run(){<br>self.P();<br>}<br>};<br>⟦Q⟧} | プロセス P，Q を並列に実行 |
| M.P | exec(⟦M⟧); ⟦P⟧ | M を実行後，P を実行 |
| (vn)P | Name n = A.createName(); ⟦P⟧<br>A.createName(String name) | 名前の生成<br>サーバの公開名を生成 |
| M[P] | ⟦M⟧.createAmbient();<br>a.run(⟦P⟧); | アンビエントの生成 |
| ⟨M⟩ | send(⟦M⟧); | メッセージ送信 |
| (x)P | Object x = recv(); ⟦P⟧ | メーッセージ受信 |
| rec X.P | while(true){⟦P⟧} | 繰返し |

### 3.2 API

この節では，アンビエント計算を Java で記述するための具体的な API を説明する（**表 1**，**表 2**）．本フレームワークでは，主に 4 つのクラスを用いる．

- アンビエントの名前を表す Name クラス
- アンビエントを表す Ambient クラス
- アンビエントのプロセスを表す Process クラス
- ケーパビリティを表す Capa クラス

先に述べたとおり，Ambient クラスにはサブクラスが存在し，Name クラスにもそれに対応するサブクラスが存在する．

#### 3.2.1 Ambient の定義

ユーザ定義のアンビエントクラスを記述するのに必要な API の説明をする．たとえば，$a[\overline{\text{in}}\ a.\text{open}\ b.(x)\text{print}(x) \mid \text{print}(\text{"hoge"})]$ というアンビエント計算を本フレームワークで

**表 2**　ケーパビリティ生成のための API
Table 2　API for creation of capabilities.

| Safe Ambients M | Java | 説明 |
|---|---|---|
| in $M$ | ⟦M⟧.createIn() | in M |
| $\overline{\text{in}}$ $M$ | ⟦M⟧.createCoIn() | $\overline{\text{in}}$ M |
| out $M$ | ⟦M⟧.createOut() | out M |
| $\overline{\text{out}}$ $M$ | ⟦M⟧.createCoOut() | $\overline{\text{out}}$ M |
| open $M$ | ⟦M⟧.createOpen() | open M |
| $\overline{\text{open}}$ $M$ | ⟦M⟧.createCoOpen() | $\overline{\text{open}}$ M |
| $M.N$ | ⟦M⟧.append(⟦N⟧) | ケーパビリティの結合 |
| connect $M$ | ⟦M⟧.createConnect() | サーバへの接続要求 |
| $\overline{\text{connect}}$ $M$ | ⟦M⟧.createCoConnect() | サーバの接続要求と対の動作 |
| disconnect $M$ | ⟦M⟧.createDisConnect() | サーバへの接続解除要求 |
| $\overline{\text{disconnect}}$ $M$ | ⟦M⟧.createCoDisConnect() | サーバの接続要求要求と対の動作 |
| | ⟦M⟧.createExport(namelist) | サーバの名前リストを公開 |
| | ⟦M⟧.createImport(namelist) | サーバの名前リストを取得 |

記述すると**図 2** のようになる．このコードを用いながらアンビエントオブジェクトの定義の仕方を説明していく．

```
class A extends IMAmbient { ... }
```

アンビエントは Ambient クラスのサブクラスとして記述される．例のアンビエントクラス `A` は IMAmbient のサブクラスである．

```
Name bn;
public A(Object[] args) { this.bn = (Name)args[0]; }
```

アンビエントクラスのコンストラクタは引数として，Object の配列をとる．引数には通常の Java オブジェクトに加え，ケーパビリティの生成に使うことのできる Name オブジェクトやケーパビリティそのものを渡すことができる．ここでは引数で受け取った Name オブジェクトを $b$ を表す Name 型の変数 `bn` にセットしている．

```
public void m() { ... }
```

アンビエントの初期プロセス $P$ を表すメソッドである．public であるため外部から呼び出すことができ，`A` クラスのオブジェクトを生成した直後にこのメソッドを呼び出すことで，$a[P]$ を実現する．なお，`A` 内に記述されているため，`A` 中の他の private メソッドを呼び出せる．

```
new Process(self) {
    public void run() {
        self.n1();
    }
};
```

```
class A extends IMAmbient {
   Name bn;
   public A(Object[] args) {
      this.bn = (Name)args[0];
   }
   public void m() {
      new Process(this) {     //並行プロセスの生成・起動
         public void run() {
            ((A)self).n1();
         }
      };
      new Process(this) {    //並行プロセスの生成・起動
         public void run() {
            ((A)self).n2();
         }
      };
   }
   private void n1() {
      exec(this.name.createCoIn());    //co-in a
      exec(bn.createOpen());           //open b
      String s = (String)recv();       //メッセージの受信
      System.out.println(s);
   }
   private void n2() {
      System.out.println("hoge");
   }
}
```

**図 2**　アンビエント定義の例
Fig. 2　An example of ambient definition.

アンビエント内の並行プロセスは複数 Process オブジェクトの生成により表す．Process オブジェクトの `run()` メソッド内には，プロセスが行う動作を直接記述しない．なぜなら，アンビエントクラスのインスタンス変数や，メソッドを使ってプロセスコード本体を書くようにしているからである．代わりに `self`（プロセスが属しているアンビエント）のメソッドを呼び出す．

```
private void n1() {
   exec(this.name.createCoIn());     //co-in a
   exec(bn.createOpen());           //open b
   String s = (String)recv();       //メッセージの受信
   System.out.println(s);
}
```

プロセスが行う動作をアンビエントオブジェクトが持つメソッド n1()，n2() として記述する．private メソッドとして書くことで，外から呼び出されることはなく安全性が保たれる．メソッド内で exec(c); と書くことで，ケーパビリティ c が実行される．Name オブジェクトの create*XXXX*（*XXXX* はケーパビリティの種類）メソッドを呼び出すことで，その名前に対する動作であるケーパビリティを作ることができる．たとえば，exec(bn.createOpen()); と書くことで，open $b$ が実行される．Capa オブジェクトは，通常の値として扱うことができ，たとえばメッセージとして送受信可能である．

```
send(bn.criateIn());
Capa c = (Capa)recv();
exec(c);
```

と記述すると，exec によって in $bn$ が実行される．

アンビエント計算の open 操作は，インスタンス変数とインスタンスメソッドを考慮に含めると仕様が複雑になる．たとえば，$a[\text{open } b.P \mid b[\overline{\text{open}}\ b.Q]]$ が実行されると，$a$ と $b$ は，お互いが持っているインスタンスメソッド，インスタンス変数を利用できるようになる．もし $a$ と $b$ がともにインスタンス変数 i を持っていた場合，open 動作後，どちらの i を使用するかはユーザの判断に委ねられる．同名のインスタンス変数を持っているときは，インスタンス変数の扱いを明確にするため下記のように記述する．

```
@open B
class A extends IMAmbient {
  int i = 0;
  int j;
  void m1() {
     ...
     exec(bn.createCoOpen());
     ...k...;
  }
}
@coopen
class B extends STAmbient {
  int k;
  @coopen int i = 10;
  void n1() {
    ...
    exec(bn.createCoOpen());
    ...j...;
    print(x);
  }
}
```

アンビエントの内部で open，$\overline{\text{open}}$ 動作を実行する場合，クラスを定義する直前に，内部で行われる open 動作をアノテーションで表記する（上の例では@open B，@coopen）．また，重複するインスタンス変数が存在するときは，重複するインスタンス変数の前にアノテーション@coopen を付ける．open が実行されると，@coopen が付いた側のインスタンス変数は上書きされる．上記の例だと，B の持っていた値 10 は 0 で上書きされる．

### 3.2.2　Name オブジェクト

アンビエントオブジェクトを生成するには，生成したいアンビエントの Name オブジェクトが先に必要となる．上の例のアンビエント $a$ に対して Name オブジェクトを生成する場合：

```
Name an = A.createName();
```

と書くことで，アンビエント $a$ を表す Name オブジェクト an が生成される．また前項で触れたように，Name オブジェクトはアンビエントオブジェクトの生成以外に，ケーパビリティを表す Capa オブジェクト（たとえば an.createIn()）の生成にも用いる．

前述のアンビエント $a$ のインスタンスを生成するときは：

```
A a = AName.createAmbient(bn);
```

を実行する．bn はアンビエント $b$ の名前である．$a$ の初期プロセスを実行するには：

```
a.run(new Process(this){
          public void run() {
```

```
            m();
        }
    });
```

と記述する．これにより，$a$ の `m()` が呼び出され，さらに `n1()`，`n2()` を実行するプロセスが生成される．

### 3.3　サーバとクライアントの API

SA には，$(\nu n)P$ によりスコープを制限された「ローカルな名前」と，自由変数が表す「グローバルな名前」の 2 種類の名前が存在する．前項で説明した無引数の `createName()` 呼び出しで生成されるのはローカルな名前である．ローカルな名前は，その匿名性のため，他の名前と同一と見なされることはない．一方，分散プログラムを記述するには，複数の計算機上で動作する複数のプログラムの間で同一と見なされる名前を取り扱う必要がある．そのための方法の 1 つとして，本フレームワークでは，各プログラムでグローバルな名前を生成するために，文字列を引数として受け取る別の `createName()` を用意している．

```
Name globalname = SomeAmbient.createName(<文字列>);
```

これにより生成される名前は，異なるオブジェクトどうしであっても，さらに別の計算機上で動作しているプログラム中のオブジェクトであっても，引数の文字列表現が同じである限り同一の名前として扱われる．

一方で，複数のプログラム間で名前を共有するために，必ずグローバルな名前を生成するという方法は，グローバルな名前空間を汚す量は増やしてしまう．そこで，本フレームワークではローカルな名前をプログラム間で共有するための仕組みを名前共有サーバという形で標準で用意している．たとえば，ある計算機上で：

$ns[P \mid sa[(\nu n)Q]]$

というコードを実行しているとする（ただし，$P$ 中に名前 $n$ は自由に出現しない）．$ns$ が名前共有サーバ，$sa$ が通常のアプリケーションプログラムを表す．この状態に対して，アンビエント $ca$ が外から接続してくると：

$ns[P \mid ca[R] \mid sa[(\nu n)Q]]$

となる（ただし，$R$ 中に名前 $n$ は自由に出現しない）．しかしながら，このままでは $ca$ からは $sa$ 中で定義されたローカルな名前 $n$ を参照することができない．そこで，名前共有サーバ $ns$ を介して（より正確には $P$ に書かれた協調動作コードを用いて）$Q$ から $R$ へ $n$ を受け渡すことにする．その結果，状態は：

$ns[(\nu n)(P \mid ca[R] \mid sa[Q])]$

へと遷移し，$ca$ は $n$ を使った動作を実行可能となる．また，次のコード：

$ns[P \mid (\nu n)(sa[Q] \mid sa[Q'])]$

のように，$n$ のスコープ中に複数のアンビエントが存在していた場合，$ca$ が接続し，名前が共有されると：

$ns[(\nu n)(P \mid ca[R] \mid sa[Q] \mid sa[Q'])]$

となる．このとき，$n$ を使った動作が $Q$ と $Q'$ の両方に書かれていた場合，$ca$ がどちらの $sa$ と協調動作を行うかは非決定的に選択される．なお，$P$ に書かれている名前の仲介機能は，すべて SA の機能の範囲内で実現できることに注意してほしい．

上記の名前共有機能を実現するため，本フレームワークでは IMAmbient のサブクラスである ServerAmbient クラスを用意した．すなわち，上記例の $ns$ は ServerAmbient クラスのアンビエントである．

これまでアンビエント階層に存在しなかったアンビエント（たとえば携帯端末中で動作しているアンビエント）を新たに階層に含めたい場合，ServerAmbient アンビエントへの接続，接続解除として表現する．そのための特別な connect，$\overline{\text{connect}}$ と disconnect，$\overline{\text{disconnect}}$ という新しい動作を用意する．もともとの SA では分散システム全体を単一のプログラムとして表現することしかできないため，複数のプログラムを動的に接続するという概念がない．そのような接続操作を in や out などの動作でモデル化するとかえって煩雑になってしまうため，我々のフレームワークにおいては，このような設計にしている．in，out と別の動作にしているのは，ServerAmbient への動的な接続は厳密な意味では in，out の動作とは異なるからである．たとえば，下記のように $sa1$，$sa2$ という 2 つの ServerAmbinet が存在し，$sa2$ に属している $a$ がサーバからの接続を解除しようとする場合，仮に out を用いて：

$sa1[\ sa2[\overline{\text{out}}\ sa2.Q \mid a[\text{out}\ sa2.P]]]$

と書くと，実行結果は：

$sa1[a[P] \mid sa2[Q]]$

となり，$a$ は $sa2$ から出て $sa1$ に含まれることになる．この結果はサーバからの接続解除とは意味が異なる．

クライアント $ca$ は：

```
exec(nsn.createConnect());
```

を実行することで $ns$ に接続し，以後 $ns$ の子アンビエントとして扱われるようになる（ここで，`nsn` は $ns$ を表す名前）．$ns$ はつねにクライアントからの接続要求，接続解除要求を

受け付けているが，これはユーザが明示的に書く必要はない．接続関連の動作は親クラスの SAServer クラスで定義されており，*ns* を動かしはじめた時点で自動的に並行実行される．

前述の名前共有のための機能には，`createExport()` というケーパビリティを用いる．このケーパビリティにより，Name オブジェクトの配列を公開することができる．たとえば：

```
public void s1() {
  Name[] namelist = {n};
  exec(nsn.createExport(namelist));
}
```

と *ns* に対する動作を実行することで，共有したい名前 `n` を *ns* へ接続してくるクライアントアンビエントに対して公開する．クライアント側では：

```
exec(nsn.createImport(namelist)); //namelist は Name[] 型
```

を実行することで，*ns* が公開している名前を取得し，配列 `namelist` に格納する．

## 4. プログラム例

本章では具体例を使って我々のフレームワークでどのように分散アプリケーションを記述するかを説明する．

**図 3** のように，ある LAN 内にファイルサーバがあるとする．外から持ち込んだ携帯端末を LAN に接続し，ファイルサーバにアクセスし，携帯端末が最初から持っていた画像とファイルサーバ上に存在する画像の合成処理を行い，結果加増を受け取ってから接続を解除するという動作を行う．これらの動作を行うために必要なアンビエントは以下のとおりである．

- `Net` は LAN 全体を管理するサーバアンビエントである．
- `FileServer` はファイルサーバ上のアンビエントであり，論理的には `Net` の子アンビエントにあたる．`Img` は `FileServer` の内側で画像データを保持するアンビエントである．
- `Mobile` は携帯端末上のクライアントアンビエントである．`Mobile` の子アンビエントである `Run` が `FileServer` に移動し処理を実行する．
- `Req`，`Res` はアンビエント間のやりとりを行うためのパケットアンビエントである（定義は省略）．

まず，`Mobile` は “`sa://networks.some-domain/net/`” というグローバルな名前を持った `Net` に接続し，`Mobile` と `Net` の間で名前情報がやりとりされる（**図 4**，15–17 行目）．このとき，`Net` から `Mobile` へ送信されるのはメッセージのやりとりに使う `Req`，`Res` アンビエントの名前 `reqn`，`resn` のみで，サーバ内にある `FileServer` の名前 `fsn` は送信されていない．メッセージ送信用の名前情報を得ると，`Mobile` は `FileServer` へ行く `Run` の名前 `runambn` を `Req` オブジェクトの中に入れて `Net` に送信する（図 4，18–19 行目）．

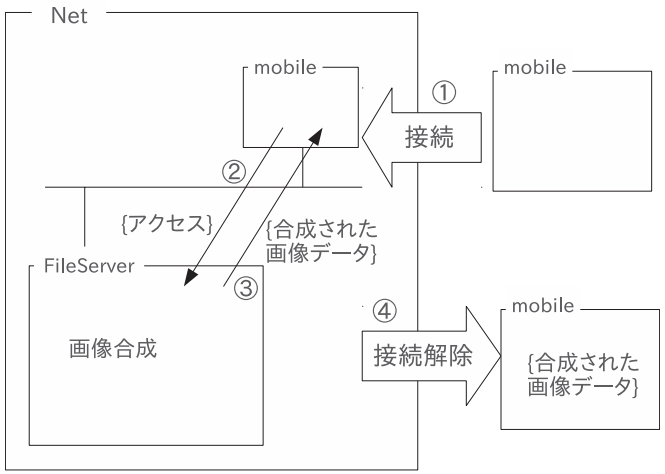


**図 3**　サンプルプログラムの動作
Fig. 3　Execution of a sample program.

同時に，`FileServer` も `Net` と名前情報のやりとりを行う．`FileServer` は `Net` に，`Net` から `FileServer` 内への移動方法を示したケーパビリティ `c` を `Req` オブジェクトの中に入れて `Net` に送信する（**図 6**，15–17 行目）．

`Net` は，`FileServer`，`Mobile` からの `Req` が届いたら，それぞれのアンビエントに必要な情報を `Res` オブジェクトとして返す（**図 5**，21–28 行目）．この場合だと，`FileServer` には `Img` が後で使用する `Run` の名前 `runambn` が伝えられ，`Mobile` には `FileSerevr` 内への移動方法（`fsn.createIn()`）が伝えられる．`Mobile` には，`FileServer` の名前 `fsn` ではなく，`FileServer` への移動方法を表すケーパビリティしか渡していない．これにより，名前の匿名性を維持できる．

`Mobile` は，受け取ったケーパビリティを `Run` に渡し，`Run` はそれを実行する（図 4，25–29 行目）．`FileServer` は受け取った `Run` の名前 `runambn` を `Img` に渡す（図 6，24–26 行目）．`Img` は `runambn` を受け取ると，`Run` への in 待ち状態になる．

`Run` は `FileServer` に入ると，`Img` を受け入れる（**図 7**，15 行目）．`Img` が入ってきた後，

```
1  class Mobile extends IMAmbient {
2     Name netn, reqn, resn;
3     public Mobile () {
4        netn = SAServer.createName("sa://networks.some-domain/net/");
5        Name mambn = Mobile.createName();
6        Mobile ma =  (Mobile)mambn.createAmbient();
7        ma.run(new Process(this) { public void run(){ mset();}});
8     }
9     public void mset() {
10       new Process(this){ public void run(){ ((Mobile)self).connect();}};
11       new Process(this){ public void run(){ ((Mobile)self).afterConnect();}};
12       new Process(this){ public void run(){ ((Mobile)self).runambs();}};
13    }
14    private void connect() {
15       exec(netn.createConnect());     //Net に接続
16       exec(netn.createImport(imports)); //名前を取得
17       /**imports から reqn,resn をセット**/
18       Req rq = reqn.createAmbient(this.name,runambn); //Req を実行
19       rq.run(new Process(this) { public void run() { reqset(); }} );
20    }
21    private void afterConnect() {
22       exec(this.name.createCoOut());       //Req が out
23       exec(this.name.createCoIn());        //Res が in
24       exec(resn.createOpen());         //Res を open
25       Capa c = (Capa)recv();           //FS 内への移動方法を取得
26       Capa itinerary = this.name.createOut().append(c); //FS への移動法を作成
27       Name runambn = Run.createName();
28       Run r =  rumambn.createAmbient(itinerary);
29       r.run(new Process(this) { public void run() { runset(); }} );
30    }
31    private void runambs() {
32       exec(this.name.createCoOut()); //Run が out
33       exec(this.name.createCoIn()); //Run が in
34       exec(netn.createDisConnent()); //Net から接続解除
35    }
36 }
```

**図 4**　Mobile アンビエント
Fig. 4　Mobile ambient.

```
1  class Net extends SAServer {
2     Name reqn, resn;
3
4     public Net() {
5         Name netn = Net.createName("sa://networks.some-domain/net/");
6         Net nt = (Net)netn.createAmbient();
7         nt.run(new Process(this) { public void run() { netset(); }});
8     }
9     public void netset(){
10       new Process(this) { public void run() {((Net) self).exname(); }};
11       new Process(this) { public void run() {((Net) self).meschange(); }};
12       new Process(this) { public void run() {((Net) self).fsrun(); }};
13    }
14    private void exname() {
15       reqn = Req.createName();
16       resn = Res.createName();
17       Name[] exports = {reqn, resn};
18       exec(netn.createExport(exports));
19    }
20    private void meschange() {
21       while(true){
22          exec(reqn.createOpen()); //Req を open
23          Object obj = recv();     //Req から必要な情報を得る
26          Res res = resn.createAmbient();
24          /**FS 宛なら，Mobile から来る Ambient の名前 runambn を res に格納**/
25          /**Mobile 宛なら FS 内への移動方法をケーパビリティとして res に格納**/
27          res.run(new Process(this) { public void run() { resset(); }});
28       }
29    }
30    public void fsrun() {
31       Name fsn = FileServer.createName();
32       FileServer fs = fsn.createAmbient();        //FS を実行
33       fs.run(new Process(this) { public void run() { fsset(); }});
34    }
35 }
```

**図 5**　Net アンビエント
Fig. 5　Net ambient.

```
1  class FileServer extends IMAmbient {
2    Name reqn, resn, netn;
3    public FileServer() {
4       netn = SAServer.createName("sa://networks.some-domain/net/");
5       Name fsn = FileServer.createName();
6       FileServer fs = (FileServer)fsn.createAmbient();
7       fs.run(new Process(this) { public void run() { fsset(); }});
8    }
9    public void fsset() {
10      new Process(this){ public void run() { ((FileServer)self).connect(); }};
11      new Process(this){ public void run() { ((FileServer)self).come(); }};
12   }
13   private void connect() {
14      exec(netn.createImport(imports)); //Net から reqn,resn を取得
15      Capa c = this.name.createIn()
16      Req rq = reqn.createAmbient(this.name, c);
17      rq.run(new Process(this) { public void run() { reqset(); }});
18   }
19   private void come() {
20      exec(this.name.createCoOut());  //Req アンビエントが out
21      exec(this.name.createCoIn());   //Res アンビエントが in
22      exec(resn.createOpen());        //Res アンビエントを open
23      Name runambn = (Name)recv();    //runambn を取得
24      Name imgambn = ImgAmbient.createName("image1"); //ImgAmb をセット
25      ImgAmbient imgamb = imgambn.createAmbient(runambn);
26      imgamb.run(new Process(this) { public void run() { imgset(); }});
27      exec(this.name.createCoIn());   //RunAmbn が in
28      exec(this.name.createCoOut());  //RunAmbn が out
29   }
30 }
```

**図 6**　FileServer アンビエント
Fig. 6　FileServer ambient.

Img のグローバルな名前 imgn を用いて Img を open する（図 7，16 行目）．ここで，Img の名前は，グローバルな名前ではあっても URI のような一意性を持った名前である必要はないことに注意してほしい．この名前は FileServer と Run の間だけで一致していれば十分だからである．Img を open 後，Img が持っている画像データ img2 と自身が持つ画像データ img1 を合成する（図 7，17 行目）．合成処理が終了すると，FileServer から Mobile へ戻る．（図 7，18 行目）．inverse() はケーパビリティの動作を逆にする関数である．たとえ

```
1  @open Img
2  class Run extends STAmbient { //FS に入るアンビエント
3     Capa itinerary;
4     Image img1 = ... ; //画像データを持つ
5     Name imgambn;
6     public Run(Object[] args) {
7        itinerary = (Capa)args[0];     //FS への移動法をセット
8        imgambn = Img.createName("image1"); //Img の名前をセット
9     }
10     public void runset() {
11        new Process(this){ public void run() { self.moves();}};
12    }
13    private void moves() {
14       exec(itinerary);               //FS へ移動
15       exec(this.name.createCoIn()); //画像アンビエントが in
16       exec(imgambn.createOpen());     //画像アンビエントを open
17       img1 =  mergeImg(img1, img2);      //画像を合成
18       exec(itinerary.inverse()); //FS から mobile へもどる
19    }
20    public Image mergeImg(Image img1, Image img2){...}
21 }
22
23 @coopen
24 class Img extends STAmbient { //画像アンビエント
25    Image img2 = ... ; //画像データ
26    Name runambn;
27    public Img(Object[] args) {
29       runambn = (Name)args[0];
30    }
31    public void imgset() {
32        new Process(this){ public void run() { self.ins();}};
33    }
34    private void ins() {
35       exec(runambn.createIn());        //Run に in
36       exec(this.name.createCoOpen());  //画像アンビエントを open
37    }
38 }
```

**図 7**　Run アンビエントと Img アンビエント
Fig. 7　Run ambient and Img ambient.

ば，out $a$.in $b$.in $c$ が与えられると，out $c$.out $b$.in $a$ を返す．`Run` が `Mobile` に入ってくると，`Mobile` は `Net` からの接続を解除する（図 4，33–34 行目）．

特に，`Run` 中の `itinerary` や `img1`，`Img` 中の `img2` を使うことで，アンビエントの移動や協調動作が簡潔に書けていることに注意して欲しい．

## 5. 実　　装

本章では本 Safe Ambients の動作を実現するためのフレームワークにおける実装方法について述べる．

### 5.1 PAN

今回 SA の動作を Java 上で実装するにあたり，PAN [7] と呼ばれる SA プロセスを実行するための抽象機械と同じ実装手法を用いる．なお，PAN およびその改良[7),9)] は，Java（あるいは OCaml）により実装された SA プロセス式のインタプリタである．Java（OCaml）コードを埋め込む機能や，Java（OCaml）コードと連携する機能も備えていないため，SA プロセス式として表されたモデルを実行することしかできず，現実のアプリケーションを開発するために用いる処理系とはいえない．

PAN では，アンビエントの動作が実行されたとき，open を除いては計算機間のアンビエントの物理的移動は生じない．たとえば，アンビエント $a$，$b$，$c$ が存在し，$c[a[\text{in } b.P] \mid b[\overline{\text{in}}\ b.Q]]$ を実行する．論理的には，in が実行されると $a$ は $b$ の中に入る．しかし，物理的に $a$ と $b$ が別々の計算機上で動作している場合，この in 動作が実行されても $a$ は $b$ のいる計算機上へは移動せず，$a$ は元の計算機上で動き続ける．open により $a$ の膜が開放されるまで $a$ は $b$ と実際にインタラクションできないため，open 動作時まで物理的な移動を遅延させ，アンビエントの移動にかかる通信コストをなるべく減らそうというアイデアである．

PAN を用いたときの，各動作に対するアンビエントの挙動は**図 8** のようになる．箱はアンビエントを表しており，箱を結ぶ線はアンビエントの親子関係を示している．$c[a[\text{in } b.P] \mid b[\overline{\text{in}}\ b.Q]]$ を実行しようとした場合，$a$ から $c$ へ {in} というメッセージが送られ，同時に $b$ から $c$ へ $\{\overline{\text{in}}\}$ というメッセージが送信される．$c$ は到着したメッセージを照合し，in と $\overline{\text{in}}$ が対応していたら，$a$ に対して {go b} というメッセージを，$b$ に対しては $\{\text{OK}\overline{\text{in}}\}$ というメッセージを送信する．$a$ は {go b} を受信したら，メッセージ中の $b$ という情報をもとに，自分の親を $c$ から $b$ へと変更する．$a$ は親の情報を更新するだけであって，アンビエント自身が別の場所へ移動しているわけではない．out 操作も同様であり親の情報が変わるだけであって，アンビエントは移動しない．これが PAN の大きな特徴である．論理的には，あるアンビエントの内側にいても，物理的には別の場所に存在することがある．

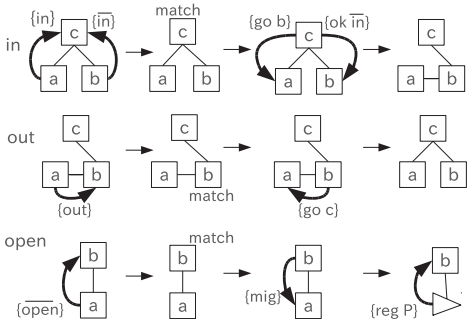


**図 8**　PAN における動作
Fig. 8　Actions in PAN.

open 操作は in，out とは少し異なる．$b[\text{open } a.Q \mid a[\overline{\text{open}}\ a.P \mid R]]$ が実行されると，まず $a$ は $b$ に対して $\{\overline{\text{open}}\}$ を送信する．$b$ は $a$ からのメッセージを受信し，対応が確認できたら $a$ に対して，メッセージ {mig} を送信する．$a$ は {mig} を受信すると，さらに {reg $P \mid R$} を $b$ に送信する．このメッセージ中のプロセス式は $a$ の継続を意味し，$b$ は {reg $P \mid R$} を受信すると，それを自身の中に入れる．これによりコードの物理的なマイグレーションが行われる．送信後，$a$ 自身はフォワーダとなる．フォワーダの説明はここでは省略する．PAN の形式的な操作的意味については参考文献 7) を参照してほしい．

### 5.2 全 体 構 成

3 章で述べたように，我々のフレームワークにおいてはアンビエントは Java オブジェクトである．アンビエント計算は，複数の JVM 上に，複数のアンビエントオブジェクトが配置されていて，JVM 間で（コードマイグレーション含む）アンビエント間の協調動作が行われていると見ることができる（**図 9**）．

各 JVM 内部の複数のアンビエントは，それぞれが並列に動作している．並列に動作させるために，1 つのアンビエントに少なくとも 1 つの Java スレッドを割り当てる必要がある．その際，アンビエントには STAmbient と IMAmbient の 2 種類が存在する．STAmbient では，in，out，open（およびそれに対応する補動作）が 2 つ以上同時に実行可能になることがないので，STAmbient 内のプロセスはまとめて 1 つの Java スレッドで実行可能であり，本フレームワークではそうしている．ただし，1 つの Java スレッドしか割り当てられてないため，あるプロセスが入出力処理でブロックすることなどを考慮に含め，現時点の実

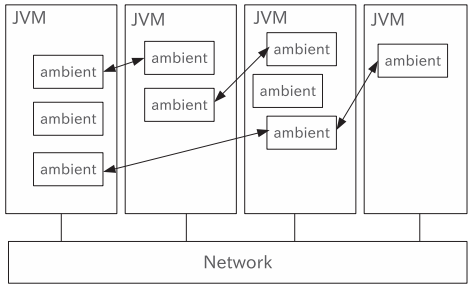


**図 9**　全体構成
Fig. 9　Structure of our framework.

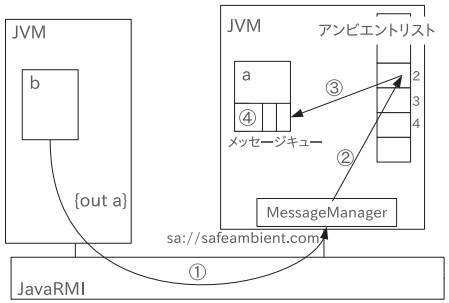


**図 10**　メッセージの送信
Fig. 10　Message sending.

装では複数のプロセスをアンビエント中のキューによって管理している．in 動作などでメッセージの受信待ちが起こると，プロセスは中断する．プロセスが中断されると，そのプロセスをキューの末尾に追加し，キューの先頭から取り出したプロセスを実行する．

一方，IMAmbient では，$\overline{\text{out}}$ と $\overline{\text{in}}$ などの動作を同時に実行する可能性があるため，1 つのプロセスに対して 1 つの Java スレッドを割り当てている．

### 5.3　PAN を Java で実現するための低水準機能

#### アンビエントの識別

PAN の手法を用いてアンビエント計算を実装するには，各アンビエントを一意に表す識別子が必要となる．たとえば，in の動作を行う際，{in} というメッセージを送ろうとするなら，送り先である親アンビエントの情報が必要であり，また，{go b} というメッセージが届いとき，$b$ が示しているアンビエントは一意に定まらなければならない．

単一の JVM 内においてアンビエントを識別するために，各アンビエントは ID を持っている．この ID は次のようにして生成される．各 JVM はアンビエントの配列を保持している．JVM 内で `createAmbient()` が実行されるたびに，作成されたアンビエントが配列に追加される．追加されたときのインデックスが，そのアンビエントの ID となる．配列で ID データを管理しているため，JVM 内のアンビエントは一意に識別される．

同じ JVM 上にいるアンビエントどうしであれば，ID さえ分かれば相手のアンビエントを特定することができるが，異なる JVM 上にいるアンビエントを特定しようとするとアンビエントの ID だけでは足りない．どの JVM に属しているのかが分からないからである．この問題を解決するために，JVM が動作しているホストの情報とネットワーク通信のために用いるポート番号を用いる．ホスト，ポート番号，アンビエントの ID の 3 つを合わせることで，他の JVM からでもアンビエントを一意に定めることができるようになる．たとえば，ある JVM がホスト “safeambient.com” 上で動作しており，ネットワーク通信に用いるポート番号が 9999，アンビエント $a$ のインデックスが 3 であるならば．任意の JVM からこのアンビエント $a$ は “`saj://safeambient.com:9999/3`” という文字列で一意に特定することができる．すべてのアンビエントは，この文字列形式で親情報を持っている（自分が最上位であれば空）．

#### メッセージのやりとり

アンビエントの動作が実行されると，親子間でメッセージのやりとりが行われる．たとえば，ある動作を実行するアンビエントオブジェクトと，それと対になる動作を動かすアンビエントオブジェクトの間でメッセージのやりとりが発生する（in の場合は，さらに両者の共通の親もメッセージのやりとりに含まれる）．別の JVM 上で動いているアンビエントオブジェクトにメッセージを送る必要もあるが，今回の実装ではそれを実現するために JavaRMI を用いている．

メッセージの送信の際には，先に述べたホスト，ポート番号，ID の組をそのままアドレスとして用いる．たとえば，$c[a[\overline{\text{out}}\ a \mid b[\text{out}\ a.Q]]]$ を実行する場合を考える（**図 10**）．図中の矢印が示すように $b$ から $a$ へ $\{\text{out}\ a\}$ が送信される．なお，本実装ではメッセージにアンビエントの Name オブジェクト $a$ を付けて送信している（理由は後述）．

$b$ は自分の親である $a$ のアドレス “`saj://safeambient.com:9999/2`” を持っているので，このアドレスに対してメッセージが送信される．すべてのメッセージには，送り主のアドレスが付与される．このアドレスは返信の際に使用される．

まず，$b$ は “safeambient.com:9999” の示す JVM へメッセージを送信する（図 10-1）．RMI の接続が成功すると，safeambient.com 側の JVM にある MessageManager が呼び出され，ID（この場合 2）が指しているアンビエントオブジェクトにメッセージが転送される（図 10-2）．

すべてのアンビエントオブジェクトは，メッセージを受信するためのメッセージキューを持っている．メッセージが届くとメッセージキューにメッセージが追加され，メッセージに対する処理を行う際にメッセージはキューから削除される．この例の場合，{out $a$} というメッセージが $a$ のキューに追加される（図 10-3）．

$a$ が $b$ に {go c} を返信するには，{out $a$} に付与していた b のアドレスを送信先としてメッセージを送信する．送信処理自体は {out $a$} と同様である．この $c$ は厳密には $c$ のアドレスのことである．$c$ は $a$ の親であるため，$a$ は $c$ のアドレスを知っている．$b$ は {go c} を受け取ると，自分の親の情報を $c$ のアドレスに置き換える．in 動作も同様の流れになる．

なお，3 章で述べたように，アンビエントの動作を実行する時点においては，必要な名前は（名前共有サーバを介するなどして）あらかじめ交換されている．

$c[a[\overline{\text{out}}\ a \mid b[\text{out}\ a.Q]]$ を実行すると，$b$ から $a$ に {out $a$} が送信される．メッセージ内の $a$ は $a$ を表す Name オブジェクトである．$a$ は {out $a$} を受信すると，自分の内部の $\overline{\text{out}}\ a$ と対になっているかどうかを確認する．確認には，「out 動作である」という情報とメッセージに付けられている Name オブジェクトを用いる．Name オブジェクトにもグローバルにユニークな ID が付与されているため，{out $a$} の $a$ の ID と $\overline{\text{out}}\ a$ の $a$ の ID が一致すれば，同じアンビエントを指していると判断でき，この 2 つの動作は対応づけられる．

**アンビエントの移動**

前述したとおり，open 動作を実行するとプロセスの物理的移動が起こる．{$\overline{\text{open}}$} などのメッセージのやりとりは，in，out と同じようにアドレスと RMI を用いてやりとりが行われる．たとえば，$a[\text{open}\ b.P \mid b[\overline{\text{open}}\ b.Q]]$ を実行するとする．

$b$ は {mig} を受け取った時点では，$a$ とは別の計算機上に存在しているとする．メッセージを受け取った後，$b$ のプロセスはいったん停止され，それらは $a$ のいる計算機上へと物理的に移した後，さらにアンビエント $a$ の中に入れられてから実行再開する．

プロセスの移動には一般的に「強い」移動と「弱い」移動の 2 種類が存在する．あるプロセスの実行中にプロセスの移動が発生した場合，移動先でそのプロセスを最初からやり直すのではなく，移動が発生した時点の状態からプロセスを再開する手法を「強い」移動性と呼ぶ．一方，実行スタックの復元を行わず，移動先でそのプロセスの実行を最初からやり直す手法を「弱い」移動性と呼ぶ．本フレームワークでは利便性のため「強い」移動性をサポートしている．「強い」移動性の実現にあたっては，今回，JavaGo[14] を下位の Java スレッドマイグレーション機能として用いることにした．ただし，本実装では，JavaGo を少し改良している．JavaGo では，コードを undock { ... }でくくることにより部分継続の範囲を指定でき，その中，あるいはそこから呼び出されるメソッドの中で go("jgo://host:port") を実行することにより，その部分継続を go 文の引数に指定した別の計算機へ移動させ実行再開させられる．これを改良し，本フレームワークでは，undock の中で特殊な URL を用いて go("saj://suspend") のように実行すると，部分継続を表す実行スタックが実行している JVM のヒープへと退避されるようにした．退避された実行スタックは，プロセスとしてアンビエント中のキューに追加される．このプロセスは，どこでも任意のタイミングで実行することができ，上の例だと，$b$ から $a$ への {reg} メッセージに含ませることもできる．

$b$ のプロセスキューに入っているプロセスの中には，$b$ への参照 self が含まれている可能性があるが，プロセスの実行再開時にキュー内のプロセスが持っている self をすべて $a$ へと置き換えることで対処している．

### 5.4　PAN の動作の実現

Pan の動作は，Ambient クラスの exec() メソッド中により実現される．exec() メソッドは，ケーパビリティを引数として受け取る．

たとえば，$c[b[\overline{\text{out}}\ \ b.Q \mid a[\text{out}\ \ b.P]]]$ を実行する際，アンビエントクラス A のあるメソッドに exec(bn.createOut()) が書かれており，同様に，B のあるメソッドには exec(bn.createCoOut()) が書かれている．

A において exec(bn.createOut()) が実行されると，自分の親 $b$ に対してメッセージを送る．一方，B では，exec(bn.createCoOut()) が実行されると，メッセージ受信待ち状態となり，子アンビエントからのメッセージを待つ．B が受信待ち状態になるより先に，A がメッセージを送信したとしても，メッセージは B が持つメッセージキューに追加されるため問題は起こらない．

B はメッセージを受け取ると，メッセージの内容 {out $b$} が，自分が行おうとしている動作 $\overline{\text{out}}\ b$ と対応しているかを照合する．マッチしているならば，子アンビエント $a$ に対して，自分の親 $c$ の情報を付与したメッセージ {go $c$} を送信する．マッチしなければマッチするメッセージが届くまで引き続きブロックする．A は {go $c$} というメッセージを $b$ から受け取ると，メッセージをもとに親情報を $c$ に書き換える．in，open の場合の動作も exec() メソッド中で同じように実装している．

### 5.5　アンビエントのマージ

$a[\text{open } b.P \mid b[\overline{\text{open}}\ b.Q]]$ が実行されると，$a$ と $b$ はお互いが持っているインスタンスメソッドおよびインスタンス変数を利用できるようになる．

このようなオブジェクトのマージ機能を Java 言語は備えていないため，本フレームワークでは Java コードへのプリプロセスを行う．たとえば：

```
class A extends IMAmbient {
  int i;
  void m1(){
     ...;
     exec(bn.createOpen);
     ...k...;
  }
}
class B extends STAmbient {
  int k;
  void n1() {
    ...;
    exec(bn.createCoOpen);
    ...i...;
 }
}
```

と記述されていたとする．A の m1 メソッドでは，B のインスタンス変数である k にアクセスしており，B の n1 メソッドは A のインスタンス変数である i にアクセスしている．

これらのインスタンス変数アクセスを可能にするために，プリプロセスの段階で A クラスと B クラスを書き換えて，AB クラス，B クラスを作成する．クラスの組合せは，3 章で述べたアノテーションを用いて判断する．ここで，AB クラスは B クラスのサブクラスである．B クラスは STAmbient のサブクラスであるため，AB クラスも STAmbient のサブクラスになる．本来であれば A クラスは IMAmbient のサブクラスである．変換後も AB クラスが IMAmbient のサブクラスとして振る舞えるよう，AB クラスは IMAmbient を委譲している．AB は B のサブクラスであるため，B のインスタンス変数にアクセス可能である．また，A のフィールドやメソッドへアクセスする B のメソッド定義は AB クラスへ移動される．上記例は以下のように変換される．

```
class AB extends B {
  int i;
  void m1(){
     ...;
     exec(bn.createOpen);
     ...k...;
  }
  void m2() {
    ...i...;
  }
}
class B extends STAmbient {
  int k;
  void n1() {
    ...;
    exec(bn.createCoOpen);
 }
}
```

B において $\overline{\text{open}}$ 以後の動作は AB に移されてメソッド m2 として定義される．open $b$ が実行された後，m2 メソッドが実行される．$\overline{\text{open}}$ 以後の動作を open する側で定義することで，インスタンス変数アクセスおよびインスタンスメソッドの呼び出しを可能にしている．

もし，open される前に，相手のインスタンス変数 i にアクセスしていると，自分のインスタンス変数には i が存在していないため，ランタイムエラーとなる．

## 6. 関 連 研 究

Aglets[10]，Voyager[12]，Tracy[3] など，Java 上のモバイルエージェント用フレームワークは数多く存在するが，本章では特に，階層化された場所の概念を導入しているか，あるいは分散プロセス計算に基づいている Java フレームワークのいくつかを簡単に紹介し，我々が今回提案するライブラリとの比較を行う．

Mobile Ambients[4]（MA）を提案した Cardelli 自身による MA の Java による実装に関

する論文がある[5]．しかし，その論文では，単一 JVM 上での synchronized ロック機構を用いた実現方法についてのみ提案されており，分散アプリケーションを構築するためのフレームワークとはいえない．

文献 13) では，アンビエント計算と同様に階層化された場所の概念を持ち，かつ場所間の通信に π 計算のチャネル通信を利用する分散計算モデル $\delta\pi$-calculus を考案し，さらに，$\delta\pi$-calculus に基づく分散アプリケーションを Java プログラムとして記述するためのフレームワークについても提案している．計算モデル自体は SA と似ているが，文献 13) の Java フレームワークでは計算機間の接続関係を場所の階層関係として表すことはできず，各計算機はそれぞれ独立したアンビエント階層を持っている．そのため，我々が提案する計算機間の接続関係の動的な変更は行えない．また，実装においてはすべてのプロセスが必ず 1 つの Java スレッドとして実行されている点，およびコードの「強い」移動性をサポートしていない点が我々のフレームワークとは異なる．

分散 JoinJAVA 言語[15] は，階層化された場所の概念を持つ Join 計算[6] に基づく拡張 Java 言語である．分散 JoinJava 言語では，プロセス間の同期・協調動作部分は DSL（Domain Specific Language）を使って記述し，DSL プログラムの中に Java のステートメントを埋め込む記法を用意することで Java との連携を図っている．そのため，Java のオブジェクト指向実行モデルとの親和性は我々のアプローチと比べ，あまり高くないといえる．また，残念ながら実行時システムに関する情報を得られなかったため，今回，我々のライブラリとの実装手法の比較はできていない．

Klava[1] は，X-Klaim と呼ばれる高水準オブジェクト指向モバイルエージェント言語のコンパイルターゲットとして開発された Java 上のモバイルエージェントフレームワークである．Klava アプリケーションを構成する各計算機やネットワークドメインを代表するゲートウェイサーバは，NetNode クラスの Java オブジェクトで表現される．階層ネットワークトポロジは NetNode オブジェクトの木構造により表現される．親子 NetNode オブジェクト間の接続関係を動的に変更する機能も備えており，それは我々のライブラリの ServerAmbient が備える connect，*disconnect* の機能とほぼ同じである．しかし，Klava では移動コードは，別の Process クラスのオブジェクトとして表現される．また，Process オブジェクト間の通信は，各 NetNode 上に 1 つずつ用意されているタプルスペースを介して行われる．したがって，Klava のモデルは，すべてをアンビエントで統一的にモデル化しようという我々のライブラリのモデルとは大きく異なる．さらに，Klava のタプルスペースは任意のプロセスからアクセス可能な公開データ格納スペースであるため安全性の問題が生じるが，Klava では，タプルを暗号化するための専用 API を別途用意することで対処している．一方，SA では，in，out，open といった操作には必ず対となる操作が必要であるため，本質的に安全なコードを記述できるようになっている．

IMC[2] は，様々な分散プロセス計算を Java 上で実現するために設計された汎用フレームワークである．場所の管理，名前の管理，通信プロトコル，コード移動のためのモジュールを提供しており，分散プロセス計算に基づくシステムを実現するために利用可能である．しかしながら，特定の計算モデルを対象としない汎用的な設計のため，我々のフレームワークと比べると，提供される機能には比較的低水準なものが多く，IMC で直接アプリケーションプログラマを書くのはあまり容易ではない．我々のフレームワークを実現するためのバックエンドフレームワークとしての利用も考えられるが，IMC では「弱い」移動のみのコード移動機能しか提供されていない．そのため，今回の JavaGo[14] を用いた実装ほど容易には「強い」移動性を実現できないものと思われる．

## 7. ま と め

本稿では，Safe Ambients に基づく並列・分散計算のための Java フレームワークを提案した．また，複数の計算機間でアンビエントプログラムを動作させるときに生じる現実的な問題に対処できるよう，SA に対しいくつかの機能拡張を行った．さらに，プログラム例を示すことで，本フレームワークを用いて実用性の高いコードを簡潔に記述できることを示した．実装には Pan の手法を用いた．

今後の課題として，本稿のフレームワークをターゲットとした高水準言語の設計と実装があげられる．我々のフレームワークを使って直接書かれたアンビエントの動作は，SA のプロセス式と比べ可読性が劣っている．また，アンビエントの動作とは無関係な Java コードと混ざっている点も，プログラムの保守性の面からあまり良いとは思えない．ユーザはアンビエントクラスの中にアンビエントの動作そのものを実現するためのフィールド（およびメソッド）を定義するが，たとえば，アンビエントの動作とは無関係な Java アプリケーションコードを別のクラスに分離し，アンビエントの動作を実現するためのフィールドとメソッドには private 修飾子をつけることで，ある程度保守性の高いプログラム構成にすることは現時点でも可能である．しかしながら，そのような書き方をフレームワークのユーザに強制できていないため，不用意にアンビエントの動作を壊してしまう可能性は残っている．

そこで，今後，アンビエントの動作部分については SA のプロセス式をそのまま記述できるような高水準言語を設計し，今回提案したフレームワークを用いる Java コードへ変換す

る処理系を実装することを計画している．

また，バックエンドで JavaGo を用いているが，JavaGo を使わずに実装を行うことも課題である．JavaGo は古い Java（1.4.2）しかサポートをしていないためである．関連研究であげた IMC の利用が可能だと考えている．上で述べた高水準言語を実現すれば．「強い」移動性は，「弱い」移動性しか持たないフレームワークへのコンパイラとして実現できる．

**謝辞**　本研究の一部は，科研費若手研究（B）（21700029）の補助を得て行った．

## 参 考 文 献


1) Bettini, L., Loreti, M. and Pugliese, R.: An Infrastructure Language for Open Nets, *Proc. SAC, Special Track on Coordination Models, Languages and Applications*, pp.373–377 (2002).
2) Bettini, L., Nicola, R.D., Falassi, D., Lacoste, M., Lopes, L., Oliveira, L., Paulino, H. and Vasconcelos, V.: A Software Framework for Rapid Prototyping of Run-Time Systems for Mobile Calculi, *Global Computing*, LNCS, Vol.3267, pp.179–207, Springer-Verlag (2005).
3) Braun, P. and Rossak, W.: *Mobile Ambients: Basic Concepts, Mobility Models, and the Tracy Toolkit*, Elsevier (2005).
4) Cardelli, L. and Gordon, A.D.: Mobile Ambients, *FoSSaCS '98: Proc. 1st International Conference on Foundations of Software Science and Computation Structure*, pp.140–155 (1998).
5) Cardelli, L.: Mobile Ambient Synchronization, Technical Report 1997-013, Digital Systems Research (1997).
6) Fournet, C. and Gonthier, G.: The Join Calculus: A Language for Distributed Mobile Programming, *Proc. Applied Semantics Summer School* (*APPSEM*), *Caminha*, pp.268–332, Springer-Verlag (2000).
7) Giannini, P., Sangiorgi, D. and Valente, A.: Safe Ambients: Abstract machine and distributed implementation, *Science of Computer Programming*, Vol.59, pp.209–249 (2006).
8) Hirschkoff, D., Pous, D. and Sangiorgi, D.: A Correct Abstract Machine for Safe Ambients, *Coordination Models and Languages, 7th International Conference, COORDINATION 2005*, pp.17–32 (2005).
9) Hirschkoff, D., Pous, D. and Sangiorgi, D.: An Efficient Abstract Machine for Safe Ambients, *Journal of Logic and Algebraic Programming*, Vol.71, No.2, pp.114–149 (2007).
10) Lange, D.B. and Oshima, M.: Mobile agents with Java: The Aglet API, *World Wide Web*, Vol.1, pp.111–121 (1998).
11) Levi, F. and Sangiorgi, D.: Mobile Safe Ambients, *ACM Trans. Prog. Lang. Syst.*, Vol.25, pp.1–69 (2003).
12) OBJECTSPACE: JGL Version 3.1 User Guide. http://www.stanford.edu/group/coursework/docsTech/jgl/index.html
13) Phillips, A., Eisenbach, S. and Lister, D.: From Process Algebra to Java Code, *Proc. Formal Techniques for Java-like Programs* (*FTfJP'02*), *affiliated with ECOOP'02* (2002).
14) Sekiguchi, T., Masuhara, H. and Yonezawa, A.: A Simple Extension of Java Language for Controllable Transparent Migration and Its Portable Implementation, *Proc. 3rd International Conference on Coordination Languages and Models* (*COORDINATION '99*), pp.211–226 (1999).
15) 佐伯昌樹，坂部俊樹，酒井正彦，草刈圭一朗，西田直樹：分散 JoinJAVA プログラムの通信エラーに対する型判定システム，技術報告 491，IEICE (SS2005-65) (2005).


(平成 22 年 12 月 17 日受付)
(平成 23 年 3 月 29 日採録)

**岡田　翔太**

1987 年生．2010 年京都大学工学部情報学科卒業．同年同大学大学院情報学研究科通信情報システム専攻修士課程入学．プログラミング言語，アルゴリズム等に興味を持つ．

**馬谷　誠二**（正会員）

1974 年生．1999 年京都大学工学部情報学科卒業．2001 年同大学大学院情報学研究科修士課程修了．2004 年同大学院情報学研究科博士後期課程修了．同年同大学院情報学研究科産学官連携研究員．2005 年同研究科助手．2007 年より同研究科助教．博士（情報学）．プログラミング言語，並列/分散処理に興味を持つ．日本ソフトウェア科学会，ACM 各会員．

**林　　奉行**

1986 年生．2010 年京都工芸繊維大学工芸学部電子情報工学科卒業．同年京都大学大学院情報学研究科通信情報システム専攻修士課程入学．プログラミング言語，並列/分散処理等に興味を持つ．

**八杉　昌宏**（正会員）

1967 年生．1989 年東京大学工学部電子工学科卒業．1991 年同大学大学院電気工学専攻修士課程修了．1994 年同大学院理学系研究科情報科学専攻博士課程修了．1993〜1995 年日本学術振興会特別研究員（東京大学，マンチェスター大学）．1995 年神戸大学工学部助手．1998 年京都大学大学院情報学研究科通信情報システム専攻講師．2003 年同大学助教授．2007 年より同大学准教授．博士（理学）．1998〜2001 年科学技術振興事業団さきがけ研究 21 研究員．並列処理，言語処理系等に興味を持つ．日本ソフトウェア科学会，ACM 各会員．平成 21 年度情報処理学会論文誌プログラミング優秀論文賞受賞．

**湯淺　太一**（フェロー）

1952 年神戸生．1977 年京都大学理学部卒業．1982 年同大学大学院理学研究科博士課程修了．同年京都大学数理解析研究所助手．1987 年豊橋技術科学大学講師．1988 年同大学助教授，1995 年同大学教授，1996 年京都大学大学院工学研究科情報工学専攻教授．1998 年同大学院情報学研究科通信情報システム専攻教授となり現在に至る．理学博士．記号処理およびプログラミング言語処理系に興味を持っている．著書『Common Lisp 入門』（共著），『C 言語によるプログラミング入門』，『コンパイラ』ほか．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．