データ構造の変更に対応したソフトウェアの動的更新手法

佐伯 智之* 河口 信夫‡ 稲垣 康善†

*名古屋大学工学部電気電子情報工学科
†名古屋大学大学院工学研究科
‡名古屋大学情報連携基盤センター

概要

ユビキタス環境において，身の回りの各機器にインストールされているソフトウェアのアップデート作業を簡単化するために，ソフトウェアの動的な更新を行うシステムが望まれる．ソフトウェアの動的な更新とは，動作しているソフトウェアのコードを，それがその時点でもつ内部変数などのデータを保持したまま，更新されたコードへと移行させることである．我々は，モバイルエージェントに基づいて動的更新を実現する手法を提案してきた．この動的な更新を行う際，更新前後のソフトウェア間で，その実行状態のデータ構造が異なる場合に，うまく更新されないという問題がある．本稿では，そのような場合にも，コードだけでなくデータも更新することで，正しくソフトウェアの動的更新を行う手法を提案する．また，その手法に基づいた動的更新システムを，モバイルエージェントシステム cogma 上に実装し，実際にエージェントに対する動的更新の有効性を確認した．

1 はじめに

近年，情報機器の小型・軽量化，および低価格化により，携帯端末や情報家電が急速に普及しつつある．それに伴い，小型の計算機が埋め込まれた機器がいたるところに存在して人の生活を支援する，ユビキタスコンピューティング環境の実現が期待されている．そのような環境においては，多くの情報機器が身の回りのさまざまな場所に存在するため，新しい機能の追加や，不具合の改修のたびにおこなわれるソフトウェアのアップデート作業を，一つ一つの機器に対して人手で行うことは煩雑であり，多大なコストが必要となることが予想される．また，そのような環境で動作しているソフトウェアは，アップデートの際に機器の再起動などの必要がないことが望ましい．

このように，ユビキタス環境では，ネットワーク環境内の各機器で動作しているソフトウェアを，それが動作中にもつ内部変数などのデータを保持したまま，機器を再起動することなく更新できるようなシステムが望まれる．

我々は，モバイルエージェントに基づいてソフトウェアの動的な更新を実現する手法を提案してきた [1][2]．ソフトウェアの動的な更新とは，動作中のソフトウェアのコードを，それが動作中に持つ内部変数などのデータを保持したまま，新たなコードへと更新することである．このような動的更新を行う際，更新前後のソフトウェア間で，その実行状態のデータ構造が異なる場合に正しく更新されないという問題があった．本稿では，コードの更新に合わせてデータも更新することにより，この問題に対応できる動的更新手法を提案する．このデータ変換の手法は，動的更新のみでなく，異なるバージョンのソフトウェアが混在する環境において，それらが互換性のないデータをやりとりする場合にも利用することができる．また，この手法に基づいて動的な更新を行うシステムを実装し，実際に作成したソフトウェアに対して動的更新を行い，本手法の実現可能性および有効性を示した．

2　動的更新における問題

ソフトウェアの動的な更新とは，動作しているソフトウェアのコードを，それが動作中にもつ内部変数などのデータ（実行状態データ）を保持したまま，更新されたコードへと移行させることである．以下に，動的な更新を実現する際の問題を挙げる．

2.1　コードの互換性

動的な更新を行うにあたって，まず問題となるのが，コードの互換性である．ここで，コードの互換性とは，ソフトウェア間でデータがうまくやりとりできるかどうかをいう．更新前後のコード間に互換性がなければ，更新の際に保持したデータが，更新後のコードに適合せず，更新実行時にエラーが起きて更新できなくなったり，正しい更新がなされないといったことが起きる．更新前のバージョンのコードと，更新後のバージョンのコードとの互換性を保つことにより，前述のような問題を回避するには，コードの更新をある程度の制限の中で行う，といったことが必要となる．それは，例えば，データ構造を変更しないような更新である．しかし，そのような制限あると，データ構造の変更を伴うような，新たな機能の追加などができず，ソフトウェアの満足な更新はできないと考えられる．

また，このような互換性の問題は，動的更新時のみに起こるわけではなく，複数のバージョンのソフトウェアが混在する環境においても起こる．例えば，更新されたソフトウェアと，何らかの理由で更新されていない古いバージョンのソフトウェアとがデータのやりとりをする際に，それらのソフトウェア間に，やりとりをするデータの互換性がなければ，正しく動作しないということが起こりうる．

そこで，コードの互換性がない場合にも，更新がおこなえ，複数のバージョンが混在した環境におけるソフトウェア間のやりとりにも対応できるような手法を考える必要がある．

2.2　コードの更新タイミング

ソフトウェアが動作している間にその更新をおこなう場合，その更新タイミングが問題となる．それは，更新のタイミングによって，保持すべきデータが異なるためである．例えば，コード中の今まさに実行している部分を更新するには，インスタンス変数などのヒープ領域内の情報に加え，実行中メソッドや関数のローカル変数などのスタック領域内の情報，プログラムカウンタなどを実行状態として保持する必要があり，そのような更新は一般に困難である．このように，ソフトウェアの動的更新では，その更新タイミングを考慮する必要がある．

3　動的更新の手法

本手法では，動的な更新をモバイルエージェントに基づいて行う．以下では，まずモバイルエージェントの特徴について述べ，次に，モバイルエージェントを利用した動的更新手法について述べる．

3.1　モバイルエージェント

モバイルエージェントは，その実行状態を保持したまま自律的に計算機間を移動し，移動先の計算機上で処理を継続するソフトウェアである．モバイルエージェントシステムでは，エージェントがノード間を移動する際の，実行状態の管理がされているが，その実行状態の範囲についてはシステムによって異なる [3].

- Weak Migration
  プログラムコードに加えて，インスタンス変数などのヒープ領域内の情報を実行状態として転送する．Java 言語の JDK1.1 以上で提供される直列化機構 (Serialization) はこの方式を容易に実現することが可能であるため，Java 言語によるモバイルエージェントシステムの多くがこの方式を採用している．

- Strong Migration
  Weak Migration 方式の範囲に加えて，実行中メソッドや関数のローカル変数などのスタック領域内の情報，プログラムカウンタを転送する．ただし，多重化された実行スレッドについては移動対象にならないことが多い．

本手法では，Weak Migration 方式のモバイルエージェントシステムを用いている．その方式では，エージェントの移動が常に可能なわけではなく，特定の状態においてのみ可能となる．この状態は，実行状態がエージェントのコードに依存していない状態であり，エージェント

に対する動的な更新が可能な状態であると考えられる．よって，動的な更新はエージェントの移動のタイミングで行うことができ，このエージェントの移動はモバイルエージェントシステムが管理しているため，ソフトウェア側で動的更新のタイミングについて考慮する必要がなくなる．

3.2 動的更新

前述したとおり，本手法では，動的更新をモバイルエージェントに基づいておこなう．モバイルエージェントは，プログラムコードと，実行状態などのデータから構成されているが，これらを一旦，通信可能なデータ形式に変換（シリアライズ）してから，ネットワーク上を移動し，移動先に到着したときに，逆の変換（デシリアライズ）をおこない，エージェントを復元する．動的更新は，このエージェントの移動を，同一ホスト上で行うことで実現する．すなわち，単純なコードのみの更新であるなら，コードのみを更新したものと入れ換え，データはそのままシリアライズし，同一ホスト上でデシリアライズすれば，更新されたコードからなるエージェントが更新前の実行状態を保持したまま復元され，動的な更新が実現できる（図 1）．

3.3 データ構造の変更

前節の方法では，更新前後のコードの互換性問題について考慮していない．よって，更新前のソフトウェアと更新後のソフトウェアとの間で，その実行状態のデータ構造が異なっていたりする場合，更新前のデータが更新後のコードに適合せず，ソフトウェアの更新を正しく行うことができない．この問題を回避するには，コードの更新をある程度の制限内で行えばよいが，それではソフトウェアの満足な更新はできない．

そこで本手法では，そのような互換性のないコードへの更新をおこなう場合については，データ変換を伴った動的更新を行う．その更新手順を以下に示す．

1. コードを更新したものと入れ替える
2. エージェントをシリアライズする
3. シリアライズされたエージェントのデータを，更新先コードに適合するように変換する
4. 変換されたデータをデシリアライズし，エージェントを復元する

このように，コードのみを更新するのではなく，データもコードの更新に合わせて更新することにより，更新前後のソフトウェア間でデータの構造が異なっていても，正しい更新を実現する（図 2）．

また，このデータ変換は，複数のバージョンのソフトウェアが混在する環境での応用も考えられる．動作している機器の実行環境の制約などの何らかの理由により，新しいバージョンのソフトウェアと，バージョンアップされていないソフトウェアとが混在しており，それらの間でやりとりされるデータの構造が異なる場合を考える．この場合，データの変換を行うことによって，異なるバージョンのソフトウェア間でデータのやりとりが可能になる．例えば，新しいバージョンのソフトウェアから，古いバージョンのものへとデータを送る際に，そのデータを相手バージョンに合わせた変換を行う．この変換処理

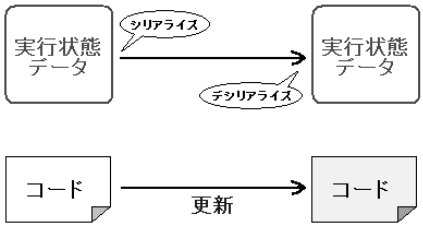

図 1: モバイルエージェントの動的更新

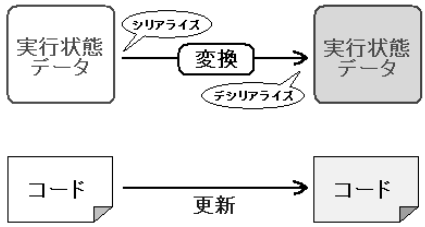

図 2: データ構造の変更に対応した動的更新

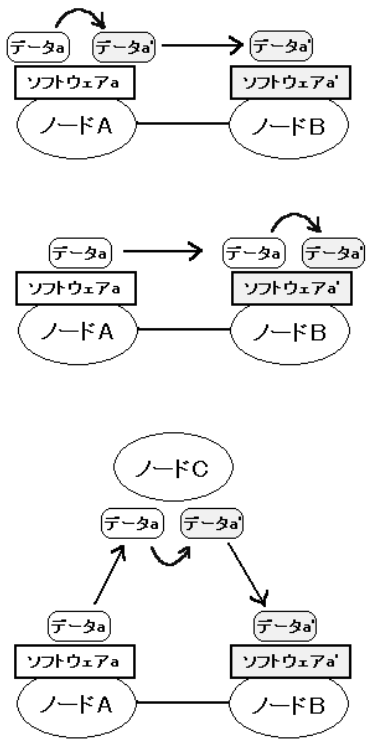

図 3: データ変換を行う場所の種類

を行う場所は，(1) データを送るノードで行う，(2) 送られたノードで行う，(3) やりとりをしているノード以外のノードで行う，の 3 通りが考えられる（図 3）．例えば，ソフトウェアが動作している機器に，メモリの制限があるなどして，余分な処理が行えない場合は，3 番目の方法を用いて，データ変換処理を他へ委託することもできる．このように，ソフトウェアが動作している機器の実行環境に応じて，データ変換が行える．

4 実装

前章で述べた手法に基づいて，動的更新を行うシステムを，我々が開発しているモバイルエージェントシステム cogma[4] を利用して実現した．

4.1 cogma

cogma(COoperative Gadgets for Mobile Appliances) は，ノートパソコンや PDA，携帯電話などの携帯端末だけでなく，家庭やオフィスの情報機器においても，無線通信などにより事前の設定なしに，また，特定のサーバを必要とせずに利用できるアドホックネットワークを構築するミドルウェアである．このシステムでは，Codget と呼ばれるエージェントプログラムがネットワーク上を移動して機器間の連携を行うことができる．また，Java で実装されているため，様々なプラットフォーム上で動作することができる．

4.2 動的更新の実現

コードの動的な更新は次のようにして行う．まず，現在実行されているエージェントをシリアライズし，次に，更新されたコードを，新しく作ったクラスローダで読み込む．これは，そのままのクラスローダで同じ名前のクラスファイルを読み込むと，以前と同じコードを使ってクラスがロードされるためである．その後，元のエージェントの実行状態データを用いてデシリアライズを行い，エージェントを復元する．これで，コードの動的な更新が行える．

実行状態データの更新（データ変換）も行わなければならないときには，元のエージェントの実行状態データをそのままデシリアライズするのではなく，更新後のコードに適合するようにデータを変換してからデシリアライズし，エージェントを復元する．

4.3 データ変換の実現

データの変換は次のようにして行う．エージェントのデータは，クラスとして記述されたエージェントのオブジェクトである．このオブジェクトの内部データに変更を加えれば，実行状態データの変換ができる．そこで，本手法では，このオブジェクトを，データとして扱いやすい形式である XML[5] に変換する．その変換には JSX(Java Serialization to XML)[6] を利用する．JSX は，Java のオブジェクトを，XML 形式のデータにシリアライズし，また，逆にそれをオブジェクトにデシリアライズする API を提供する．本手法では，XML に変換された実行状態データに対し，XSLT(XSL Transformations)[7] を用いて，フィールド値の追加や変更など，データの互換性を保つための変換を行う．XSLT は，任意の XML 文

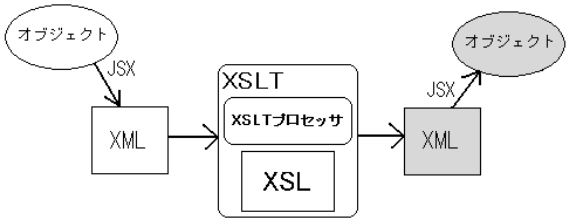

図 4: データ変換

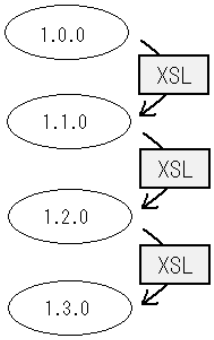

図 5: 離れたバージョンへの更新

書を読み込んで，それを加工して出力する簡易なスクリプト言語である．XSLT によりデータを加工した後，再び JSX を利用して，XML データをオブジェクトへとデシリアライズし，エージェントの復元を行う（図 4）．

このように，実行状態データを XML 化し，その後の変換に XSLT を用いることにより，内部変数の値変更や，変数追加・削除といったデータ構造の変換を，スクリプトで宣言的に記述することができる．また，データ変換処理を，更新対象となるソフトウェアのプログラムコード内に記述する必要がない．

また，データの互換性がないバージョンへと更新するためには，XSLT でデータ変換を行うための処理を記述した XSL ファイルが必要となるが，用意するのは隣接バージョン間のものだけとする．例えば，バージョン 1.0.0 からバージョン 1.1.0，1.2.0 をとばしてバージョン 1.3.0 へと更新したい場合には，まず，バージョン 1.0.0 から 1.1.0 へと更新するための XSL ファイルを用いてデータを変換し，次にそのデータを 1.1.0 から 1.2.0 へ更新するための XSL ファイルに，続いて 1.2.0 から 1.3.0 へ更新するための XSL ファイルに適用することによってデータの変換を行う (図 5).

4.4 事例紹介：図形描画エージェントの動的更新

今回，動的更新の実現可能性を示すために，簡単な図形描画エージェントを作成した．この図形描画エージェントの作成過程の 3 つのバージョンを用いて，動的な更新を行う．初めのバージョンは，単純に，直線と長方形の描画機能，描いた図形の選択機能をもつだけである（これをバージョン 1.0.0 とする）．描いた直線および長方形の移動機能をバージョン 1.0.0 に追加したものである（これをバージョン 1.1.0 とする）．そして 3 つ目のバージョンは，描いた直線および長方形の回転機能をバージョン 1.1.0 に追加したものである（これをバージョン 1.2.0 とする）．バージョン 1.2.0 では，回転の機能を付け加えるにあたって，長方形を表す Rect クラスのフィールドに，新しく，その長方形の回転角を表す変数 rotation を追加した（表 1）．よって，バージョン 1.1.0 とバージョン 1.2.0 との間で，データ構造に違いが生じており，動的更新時に実行状態データをそのまま移行したのでは，正しい更新は行われない．そこで本手法を用いる．

まず，バージョン 1.0.0 からバージョン 1.2.0 への動的更新を考える．バージョン 1.0.0 の XML データの一部を以下に示す.

```
<java.util.Vector alias-ID="3" obj-name="figV_">
  <org.cogma.codget.FigDrawer_-Rect  alias-ID="4"
   x="45" y="220" width="118" height="88"/>
  <org.cogma.codget.FigDrawer_-Rect  alias-ID="5"
   x="178" y="83" width="73" height="66"/>
  <org.cogma.codget.FigDrawer_-Rect  alias-ID="6"
   x="262" y="261" width="99" height="72"/>
  <org.cogma.codget.FigDrawer_-Line  alias-ID="7"
   x1="37" y1="55" x2="102" y2="152"/>
  <org.cogma.codget.FigDrawer_-Line  alias-ID="8"
   x1="319" y1="192" x2="366" y2="57"/>
</java.util.Vector>
```

バージョン 1.0.0 から 1.2.0 への更新では，このバージョン 1.0.0 のデータに，長方形の回転角度を表すフィールド rotation の追加を行う．こうしたデータ変換処理を記述した XSL ファイルは，バージョン 1.1.0 からバージョン 1.2.0 への更新のためのものであるが，バージョン 1.0.0 と 1.1.0 との間には互換性があるため，1.0.0 から 1.2.0 への更新にも利用できる．

表 1: 各バージョンの Rect クラスの内容

	名前	説明	1.0.0	1.1.0	1.2.0
フィールド	x	左上頂点の x 座標	○	○	○
	y	左上頂点の y 座標	○	○	○
	width	横幅	○	○	○
	height	縦幅	○	○	○
	rotation	回転角度			○
メソッド	move	移動		○	○
	rotate	回転			○

実際にこの動的更新を行った様子を，図 6 および図 7 に示す．データ変換の際に用いた XSL ファイルの一部を以下に示す．

```
<xsl:template match="org.cogma.codget.FigDrawer//
 org.cogma.codget.FigDrawer_-Rect">
 <xsl:copy>
  <xsl:copy-of select="@*"/>
  <xsl:attribute name="rotation">20</xsl:attribute>
  <xsl:apply-templates/>
 </xsl:copy>
</xsl:template>
```

ここでは，データの変換が正しく行われていることを示すため，rotation の追加の際，あえてその値を 20 としている．XSLT によって変換された XML データの一部を以下に示す．

```
<java.util.Vector alias-ID="3" obj-name="figV_">
 <org.cogma.codget.FigDrawer_-Rect alias-ID="4"
  x="45" y="220" width="118" height="88" rotation="20"/>
 <org.cogma.codget.FigDrawer_-Rect alias-ID="5"
  x="178" y="83" width="73" height="66" rotation="20"/>
 <org.cogma.codget.FigDrawer_-Rect alias-ID="6"
  x="262" y="261" width="99" height="72" rotation="20"/>
 <org.cogma.codget.FigDrawer_-Line alias-ID="7"
  x1="37" y1="55" x2="102" y2="152"/>
 <org.cogma.codget.FigDrawer_-Line alias-ID="8"
  x1="319" y1="192" x2="366" y2="57"/>
</java.util.Vector>
```

変換後のデータを見ると，Rect に rotation が追加されていることがわかる．また，図 7 で，描かれた全ての長方形が 20 度回転しており，データの変換がエージェントの更新に反映されていることがわかる．

次に，バージョン 1.2.0 からバージョン 1.1.0 へ更新する場合のデータ変換について考える．ここでは，バージョン 1.2.0 を使って回転させられた長方形が，バージョン 1.1.0 でも外見的に変わらないようにする，という方針に基づいて行う．そのために，1.2.0 におけるひとつの回転した長方形を，1.1.0 では 4 本の直線で表すこととする．このような変換処理を記述した XSL ファイルをバージョン 1.2.0 から 1.1.0 へと変換するためのものとする．その XSL ファイルの一部を以下に示す．

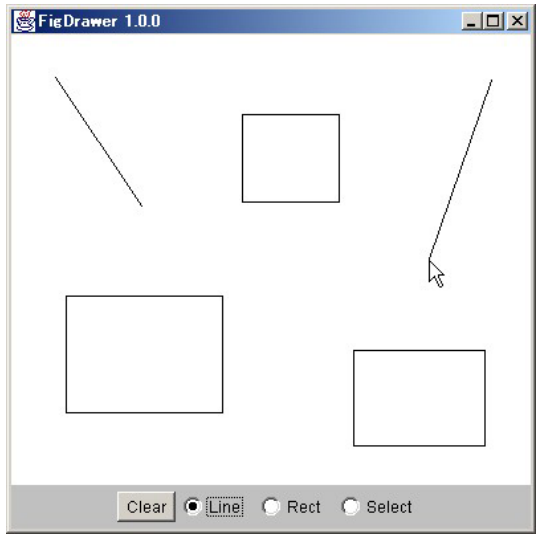

図 6: 1.0.0 から 1.2.0 への更新前

```
<xsl:template match="org.cogma.codget.FigDrawer//
 org.cogma.codget.FigDrawer_-Rect">
 <xsl:if test="@rotation[.='0']">
  <xsl:copy>
   <xsl:copy-of select="@*"/>
   <xsl:apply-templates/>
  </xsl:copy>
 </xsl:if>
 <xsl:if test="@rotation[.!='0']">
  <xsl:element name="org.cogma.codget.FigDrawer_-Line">
   <xsl:attribute name="x1">
    <xsl:value-of select="result:getRX(number(@x),number(@y),
     number(@width),number(@height),number(@rotation),0)"/>
   </xsl:attribute>
   <xsl:attribute name="y1">
    <xsl:value-of select="result:getRY(number(@x),number(@y),
     number(@width),number(@height),number(@rotation),0)"/>
   </xsl:attribute>
   <xsl:attribute name="x2">
```

```
    <xsl:value-of select="result:getRX(number(@x),number(@y),
     number(@width),number(@height),number(@rotation),1)"/>
   </xsl:attribute>
   <xsl:attribute name="y2">
    <xsl:value-of select="result:getRY(number(@x),number(@y),
     number(@width),number(@height),number(@rotation),1)"/>
   </xsl:attribute>
  </xsl:element>
          .
          .
          .
```

実際に，図 7 の状態のまま，1.1.0 へと図形描画エージェントを更新した場合の図を図 8 に示す．回転した長方形が，４本の直線から成るように変換されていることがわかる．

以上から，単純なコードの動的更新に加えて，実行状態データを変換することにより，コードの互換性がないバージョンのソフトウェアへの動的更新も行えることを示すことができた．

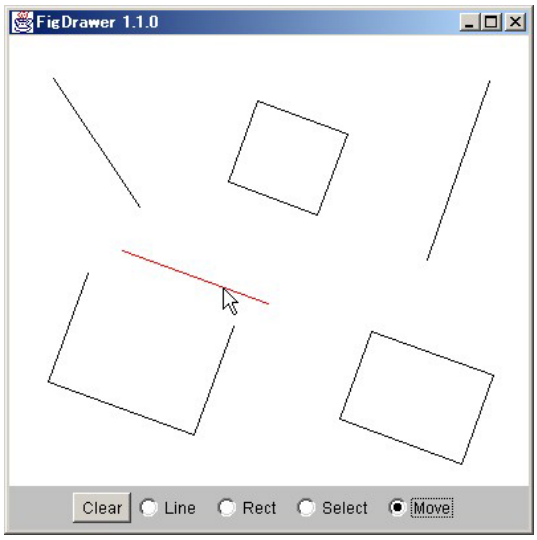

図 8: 1.2.0 から 1.1.0 への更新後

5 関連研究

ソフトウェアを動作中に更新することについての研究はいくつか存在する．しかし，それらは，本研究で目的としている，ユビキタス環境においてさまざまな場所に存在する各機器上で動作しているソフトウェアを動的に更新する，ということは想定していない．

[8] では，プロセスとして走行しているプログラムの一部分を変更する方法として，(1) サービスプログラムが，入れ替えの契機を意識しなくてよい，(2) 入れ替え対象のプログラム部分が，さらに入れ替え対象でない別プログラム部分を呼び出している場合にも，プログラムの入れ替えを可能にする，という２つの特徴をもつ方法が提案されている．これは，プロセス走行中での動的更新を考えているため，OS の機能を用いた複雑な処理を必要とし，また，入れ替え内容の自由度に制限があり，入れ替えるプログラム部分の作成が困難となっている．しかし，近年，その入れ替え条件の緩和法が提案されている [9]．

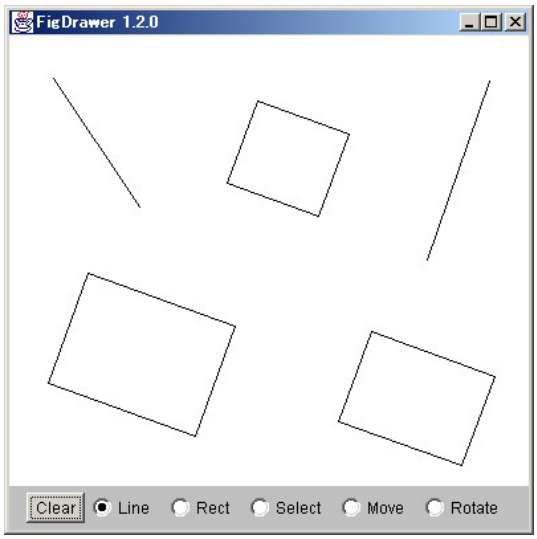

図 7: 1.0.0 から 1.2.0 への更新後

[10] では，ソフトウェアの動的な更新をサポートする，大規模な永続的分散システムの構築手法を提案している．そこでは，システムは zone と呼ばれる複数プロセスの一つの論理的集合ごとに細分され，その zone を更新の単位としている．動的更新の際のデータ変換も考えられているが，変換のためのクラスを Java で記述する必要がある．また，更新の行われた zone(のオブジェクト) が更新の行われていない他の zone(のオブジェクト) とやりとりを行う際，このシステムでは，別途 Change Absorber と呼ばれる，更新前後のオブジェクト間のインタフェースの相違を吸収するための機構が用意され，そのためのコードが必要となる．これによって，zone の外側のオブジェクトが更新に影響を受けないことを保証している．本稿で提案した手法では，動的更新時のデータ変換に用いる XSL ファイルを，そのまま他ノードのコード互換性のないバージョンのソフトウェアとのデータのやりとりに用いることができる．

[11] では，動作中のプログラムに対するバージョン変更をモデル化する形式的な枠組みについて述べられている．そこでは，動作中プログラムのバージョン変更を次のように定義している．プログラム A を実行しているプロセス P が時刻 t に状態 s で停止され，P のコードである A が A' に置き換えられ，その状態として，停止した状態 s が，状態をマッピングする S を利用して S(s) にマッピングされ，P が再開される．またこのとき，P が有限時間内に A' の到達可能状態 (A' を初期の状態から実行するプロセスが有限時間内に到達できる状態) に到達することが保証されれば，その動的変更は妥当である，と定義している．そして，この妥当性を自動的に保証するには，ユーザの補助が必要となると述べている．本稿におけるシステムでは，動的更新のタイミングが，モバイルエージェントシステムの管理のもとでエージェントの移動タイミングに制限されており，プロセスレベルでの更新は考えていない．

6 まとめ

本稿では，モバイルエージェントに基づき，データ構造の変更にも対応した，ソフトウェアの動的更新手法を提案した．また，実際に，モバイルエージェントシステム cogma 上で，その手法に基づいて動的更新を行うシステムを実現した．モバイルエージェントは，コードと，内部変数などの実行状態データからなる．このコードだけを更新して，同じノード上でエージェントを移動させることによって，エージェントの動的更新が可能となり，実行状態データに互換性がないバージョンへと更新する際には，そのデータを変換することによって，正しい更新を行うことができることを示した．また，このデータ変換手法を利用して，複数のバージョンが混在した環境でのソフトウェアの動作も行えると考えられる．

今後の課題としては，他のソフトウェアと連携して動作しているソフトウェアに対して動的な更新を行うために，ソフトウェアの依存関係を解消しながら更新する手法の検討や，データ変換のために必要となる XSL ファイルの半自動生成を行う手法の検討などが挙げられる．

参考文献

[1] 河口信夫，外山勝彦，稲垣康善：モバイルエージェントに基づく動的拡張可能なソフトウェアシステム，情報処理学会夏のプログラミング・シンポジウム，pp.71-78 (1999).

[2] 植田智，河口信夫，稲垣康善：モバイルエージェントシステムにおけるオンデマンドコード移送とバージョン管理，情報処理学会第６４回全国大会，pp.509-510 (2002).

[3] 佐藤一郎：モバイルエージェントの動向，人工知能学会誌 Vol.14 No.4，pp.22-29 (1999).

[4] 河口信夫，稲垣康善： cogma:動的ネットワーク環境における組み込み機器間の連携用ミドルウェア，情報処理学会コンピュータシステム・シンポジウム，pp.1-8 (2001).

[5] XML(Extensible Markup Language)1.0(Second Edition)：
(http://www.w3.org/TR/REC-xml)

[6] JSX(Java Serialization to XML)：
(http://www.csse.monash.edu.au/~bren/JSX/)

[7] XSLT(XSL Transformations) Version1.0：
(http://www.w3.org/TR/xslt)

[8] 谷口秀夫，伊藤健一，牛島和夫：プロセス走行時におけるプログラムの部分入れ替え法，電子情報通信学会論文誌，Vol.J78-D-I，No.5，pp.492-499 (1995)

[9] 山本淳，谷口秀夫：実行中プログラムの部分入れ替え法における入替え条件の緩和，電子情報通信学会論文誌,Vol.J85-D-I, No.8, pp.713-724 (2002).

[10] Huw Evans, Peter Dickman： Zones, Contracts and Absorbing Change: An Approach to Software Evolution, OOPSLA '99, SIGPLAN Notices vol.34, pp.415-434 (1999)

[11] Deepak Gupta, Pankaj Jalote, Gautam Barua：A Formal Framework for On-line Software Version Change, TSE 22(2), pp.120-131 (1996)