ソフトウェア工学 119－7
（1998．5．15）

動的バージョン管理に基づく実行中のソフトウェアの発展法

杉山 安洋
日本大学工学部情報工学部
sugiyama@ce.nihon-u.ac.jp

戸部 勝文
日本大学大学院工学研究科
tobe@cauchy.ce.nihon-u.ac.jp

筆者の研究室では、ソフトウェアシステムの拡張作業や修正作業を、そのシステムの実行を停止させることなく行えるメカニズムを実現する研究を行っている。我々の方式は、動的結合ライブラリの技術を発展させ、実行時でもライブラリの変更ができるようにし、ソフトウェアが発展できるようにすることである。我々の方式の特長は、実行時のライブラリの変更をライブラリのバージョン管理という方式で実現しようとしている点である。我々は、このライブラリのバージョン管理を２つの異なる方式で実現することを検討した。本稿では、まず、それらの方式の概要について述べ、各々の方式の特徴や機能の比較などを行う。次に、各々の方式を実現するために開発したシステムの概要を述べる。

A Mechanism for Runtime Software Evolution Based on Dynamic Version Management

Yasuhiro Sugiyama
Department of Computer Science
Nihon University

Masafumi Tobe
Graduate School of Engineering
Nihon University

Our goal is to develop a mechanism that allows computer programs, which are running, to evolve and change their behavior without stopping their execution. Our approach is to include a runtime version management mechanism into DLL's (Dynamic Linking Libraries) to support runtime software evolution. We have extended the traditional dynamic linking mechanism in two different ways to support the runtime version management. In this paper, We will first show the outline of our runtime version management mechanisms of DLL's. And then We will show our ways of implementing the runtime version management mechanisms of DLL's. We will conclude the paper with a summary and future works.

1. はじめに

銀行の大規模オンライン処理システムや各種のサーバシステムなどは、その一時的な停止でさえもユーザへのサービスの提供に大きな障害をもたらす。そのため、そのようなシステムの定期的保守作業は、利用者の少ない深夜などを選んで行われるが、突発的に発生した不良の修正などはこの限りではなく、多くのユーザに対して不便をかけ、システムの実行を停止して行われているのが現状である。そのため、ソフトウェアシステムの拡張作業や修正作業を、そのシステムの実行を停止させることなく行える技術の開発が望まれている。

現在広く用いられている技術の一つに動的結合ライブラリがある。動的結合ライブラリでは、ソフトウェアシステムの実行形式ファイルを作成するリンク作業において、ライブラリを実行形式ファイルに含めない。その代わり、実行形式ファイルをメモリ中にロードする際に、同時にライブラリもメモリ中にロードし、結合して使用する技術である。この技術を用いれば、ソフトウェアのリンク作業を完了したあとでも、ライブラリ関数を変更することにより、そのライブラリを使用するソフトウェアの機能を変更、追加できる。しかし、ひとたび実行が開始されてライブラリのロードが完了してしまうと、結合されたライブラリの変更はできないという欠点があった。

我々の研究室では、現在、動的結合ライブラリの技術を発展させ、実行時でもライブラリの変更ができるようにするメカニズムを実現する研究を行っている。我々の方式の特長は、実行時のライブラリの変更をライブラリのバージョン管理という方式で実現しようとしていることである。我々は、このライブラリのバージョン管理を2つの異なる方式で実現することを検討した。

本稿では、まず、それらの方式の概要について述べ、各々の方式の特徴や機能の比較などを行う。次に、各々の方式を実現するために開発したシステムの概要を述べる。

2. 実現方式

2.1. ライブラリのバージョン管理による発展メカニズム

我々の目標は、動的結合ライブラリの概念をさらに発展させ、

(1) 動的結合ライブラリ中の関数やクラスに複数のバージョンを定義することを可能とすること

(2) ロード時にはもちろん、実行開始後でも、複数用意されたバージョンからユーザが必要なバージョンを選択して実行できるようにすること

(3) 使用するライブラリ関数のバージョンの指定は、実行中でも変更できるようにし、一度ロードされたライブラリ関数でも、実行中に他のバージョンを再ロードして入れ替えることができるメカニズムを実現すること

にある。さらに、これを実現するにあたって、

(4) 既存のライブラリは、変更なく、バージョンのひとつとして利用できること

(5) ライブラリを使用するユーザプログラムの方には手を加えずに、バージョン管理機能を利用できること

を目標としている。

現行の動的結合ライブラリでは、ライブラリファイルごとにバージョンを作成することは可能であり、ライブラリファイル名の拡張子としてバージョン番号が示される。複数のバージョンを持つライブラリが存在する場合には、その最新バージョンが自動的に選択されて動的にリンクされたり、標準でリンクするバージョンが決まって

いたりする。リンカの機能のみでは、リンクされるバージョンを明示的に指定することはできないが、複数のバージョンを別々のディレクトリに入れておき、ローダが読み出すディレクトリを指定したり、ライブラリファイルのシンボリックリンクを使用することで、バージョンの明示的な指定も可能となる。しかし、これらは、あくまでローダの機能であるので、バージョンの選択ができるのは、実行を開始する時点（プログラムのロード時）までに限定される。一度実行を開始してしまうと、変更はできなくなる。従って、これらの現行の動的結合ライブラリの欠点を克服した上で、上記の5項目を満足する実行時のバージョン管理方式を実現するのが、本研究の目的である。

この実行時のライブラリのバージョンの管理機能については、いくつかの実現方式が考えられるが、本研究では、次の2つの方式で研究を行った。

2.2. 選択的動的リンク

第一の方式は、ユーザプログラムが呼び出すライブラリのバージョンの中から必要なバージョン選択し、ユーザプログラムと選択されたバージョンを直接リンクすることにより、バージョン管理を実現する方式である。本方式を、選択的動的リンク（Selective Dynamic Linking）と呼ぶ。本方式は次の3点で特徴付けられる。

(1) 選択的ローディング

まず、第一の特徴は、ローダがライブラリをロードする際に、複数あるライブラリのバージョンのうちから、ユーザが示すバージョン指定情報をもとに、必要なバージョンを選択してロードすることである。ロードされたバージョンは、ユーザプログラムとリンクされ実行される。例えば、図1は、ライブラリ中の関数Aをユーザプログラムが呼び出している場合のバージョン管理を示している。ユーザプログラムがまずローダによりプログラムの実行系に読み込まれる。ローダは、そのユーザプログラムが必要とする関数Aのバージョンをユーザが指定するバージョン情報により、選択的にプログラム実行系へロードし、実行する。

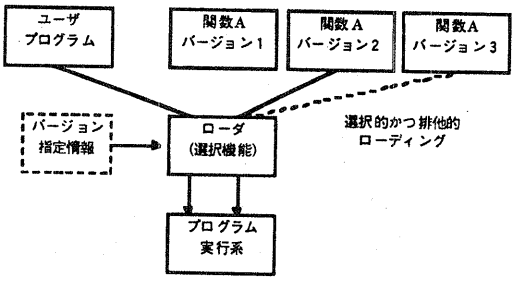

図 1：選択的動的リンクの概要

(2) 要求ベースのローディング

第二の特徴は、ライブラリのロードは、そのライブラリが必要とされるようになった段階で初めて行うことである。これは、ロードすべきバージョンが、ユーザプログラムの実行開始時には存在していない可能性があるためである。

図1では、ロードしたユーザプログラムが、関数Aを呼び出す必要が発生するまでは、関数Aをロードすることはしない。ユーザプログラムが関数Aを呼び出す段階で、初めてローダは関数Aの必要なバージョンをプログラムの実行系にロードし、そこで関数Aの実行が可能となる。

(3) 排他的ローディング

第三の特徴は、バージョンのローディングが排他的であることである。各バージョンをプログラム実行系へ直接ロードするこの方式では、ひとつの関数に複数のバージョンがあった場合でも、ユーザプログラムからは、それらはすべて同じ関数であると見える必要がある。そのため、それらのバージョンは全て同じ名前を持つことになり、ひとつの関数のバージョンのうちでプログラム実行系のメモリ中に一度にロードできるのはひとつのみとなる。従って、実行途中でロード済み

のバージョンを他のバージョンへ交換する必要が発生した場合には、ロード済みのバージョンをアンロードし、新しいバージョンをロードする必要が生じる。一度ロードが完了した関数のバージョンを入れ換えるためには、ローダあるいはプログラム実行系のシンボルテーブルの再構成などの作業が発生することは否定できない。

この方式は、Java言語の仮想マシンに機能を追加する方式で実装した。詳細は、3節で述べる。

2.3. 間接的動的リンク

第二の方式、間接的動的リンク（Indirect Dynamic Linking）は、ユーザプログラムとライブラリを直接リンクせずに、各ライブラリ関数ごとに「代理関数」(Proxy Function) を作成し、ユーザプログラムとその代理関数をリンクする方式である。代理関数は、自分がユーザプログラムに呼ばれると、ユーザプログラムに代わって、自分が代理するライブラリ関数の必要なバージョンを選択して実行し、その結果をユーザプログラムに渡す。たとえば、図2に示す通り、あるユーザプログラムがライブラリ中の関数Aを呼び出しているとする。関数Aに複数のバージョンがあった場合、ユーザプログラムが呼び出す関数Aは、実際の関数Aの処理を行うのではなく、関数Aの必要なバージョンを呼び出すためだけに使用する代理関数となる。関数Aの代理関数は、ユーザの指定するバージョンを呼び出し実行し、その実行結果をユーザプログラムへ渡す。また、代理関数が行う作業は、関数の呼び出しだけではない。呼び出す直前に、その関数をロードしたり、ロード済みのバージョンを別のバージョンへ入れ換える作業なども行う。

本方式の特長のひとつは、バージョンを定義する単位が、関数とは限定されないことである。バージョンは、オブジェクトレベルで定義できる。ユーザプログラムがライブラリ中のクラスから

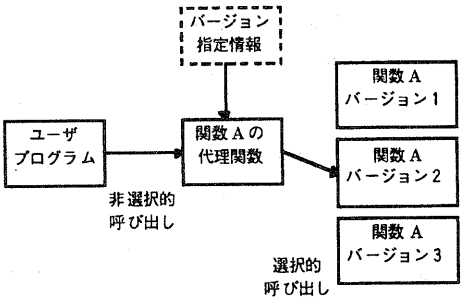

図 2:間接的動的リンク

生成されるオブジェクトを使用している場合、この方式は、オブジェクトに複数のバージョンが存在することを可能とする。ユーザプログラムは直接オブジェクトの特定のバージョンにアクセスするのではなく、そのオブジェクトの代理オブジェクトにアクセスする。代理オブジェクトは、ユーザプログラムから受け取ったアクセスの要求を、適切なバージョンへ転送し、そこで処理する。処理が完了した段階で、処理結果が代理オブジェクトを経由してユーザプログラムへ渡される。

2.4. 実現方式の比較

上記の2つの方式をいくつかの面から比較すると、次のような結果が得られる。

(1) 実行速度

第二の方式である間接的動的リンクの欠点のひとつは、関数を呼び出すごとに代理関数を経由して呼び出しが実行されるため、そのオーバヘッドが発生する点である。それに比べて、第一の方式の選択的動的リンクでは、ユーザプログラムとライブラリを直接リンクさせてしまうため、実行時のオーバヘッドは存在しない。

(2) 実現の容易さ

間接的動的リンクの大きな利点は、リンカやローダを変更しなくても実現可能であるという点である。特に、スタティックなリンク機能しか持たないリンカしか存在しない場合でも、前もっ

てすべての関数のバージョンを用意しておくという前提のもとで、バージョン管理が可能となる。ただし、実行中のソフトウェアの発展などには対応できない。一方、選択的動的リンクでは、リンカやローダの変更が必須となる。そのため、リンカやローダの変更が容易でない環境では実現が難しい。

(3) 分散オブジェクトへの対応

間接的動的リンクの利点のひとつは、RMI [6] やHORBなどの分散オブジェクトとの整合性が高く、分散オブジェクトへのバージョン管理にも応用できる点である。これは、代理関数や代理オブジェクトの概念が、HORBやRMIで用いられている方式と共通する部分が存在するからである。リモート計算機に存在するオブジェクトへアクセスするためには、ローカルな計算機上にリモートオブジェクトへのアクセスを中継するオブジェクトを用意する。中継オブジェクトは、リモートオブジェクトへのアクセスを代理として受付け、それをリモートオブジェクトへ転送する。これにより、ユーザプログラムは、あたかもローカルなオブジェクトのみをアクセスしているような感じでリモートオブジェクトをアクセスできるのである。従って、中継オブジェクトにバージョンの選択機能を用意すれば、分散環境下においてもオブジェクトのバージョンの管理が可能となる。

(4) 名前の衝突

選択的動的リンクでは、バージョン間での名前の衝突の問題をさけるために、排他的ローディングになることは、前に述べた。間接的動的リンクでも、同様な問題が発生する可能性がある。ユーザプログラムが呼び出す代理関数は、ユーザプログラムを変更しないという前提のもとでは、代理関数が代理するライブラリ関数と同じ名前を持つ必要がある。そうなった場合、もし、リンカが複数のname spaceの使用を許さない場合、代理関数と、ライブラリ関数が同時にロードできないという問題点が発生してしまう。この問題を解決するためには、代理関数を別の名前にするか、ライブラリ関数の名前を変更する必要が発生してしまい、我々の当初の目標である、ユーザプログラムおよびライブラリには変更を加えないという条件が満たされなくなってしまう。ただし、後述するように、我々がシステム開発に使用しているSolarisのリンカは、複数のname spaceを持っているので、この問題は発生していない。

3. 選択的動的リンクの実現方式

我々は、Java言語の仮想マシン [1] を改良して、選択的動的リンクを実現した。Javaの仮想マシンは、クラスをロードするクラスローダという機能を持ってはいるが、これには、クラスのバイトコードの複数のバージョンを区別したり、それらを選択的にロードする機能は持っていない。しかし、Java言語には、java.lang.ClassLoaderという抽象クラスが用意されており、そのサブクラスを作成することにより、独自のクラスローダを定義することができる。そこで、この機能を用いて実行時のバージョン管理を行う機能を持つクラスローダVCL (VersionClassLoader) を定義することとした。

VCLにおける複数のバージョンの区別は、次に示す単純な方式を採用した。Javaのバイトコードファイルは通常"class"という拡張子を持っている。ひとつのクラスのバイトコードに複数のバージョンがある場合には、それらは、ファイル名と、拡張子"class"との間に、バージョンを示す文字列をドット (.) で区切って挿入することにより実現した。バージョンを指定する文字列は、ファイル名の一部として許される任意の文字列で良く、1、2、3、1.1、1.2 などの番号の他に、バージョンの内容を示すような文字列を使用することもできる。図3の例では、クラスAのバー

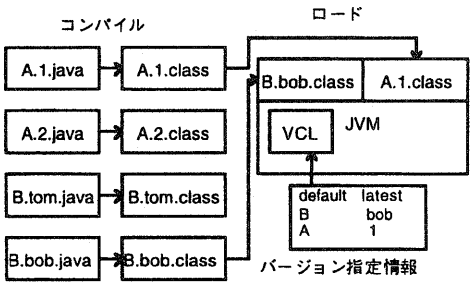

図 3：バージョン管理の基本メカニズム

ジョンは通常の1、2というバージョン番号で区別されているが、クラスBのバージョンは、開発者の名前で区別されている。

VCLに読み込ませたいバージョンを指定するためには、環境変数や、バージョン指定ファイルを使用する。環境変数による指定は、バージョン指定ファイルによる指定よりも優先するので、通常使用するバージョンをバージョン指定ファイルに記述しておき、一時的に別のバージョンが必要になった時に、環境変数により、そのバージョンを指定するなどの使いわけができる。

これらの環境変数やバージョン指定ファイルに記述するのは、図3に示すように、バージョンを指定したいクラスの名前と、そのクラスで使用したいバージョンの名称である。ここでの指定は、クラスごとにバージョン名を明示的に指定するだけではなく、明示的なバージョンの指定がないクラスに共通のバージョンを指定するdefaultなどの記述や、バージョン名称の他に、最新のバージョン(latest)や最も古いバージョン(oldest)などの指定を行うことも可能である。図3のバージョン指定ファイルでは、クラスAのバージョンは1を使用し、クラスBのバージョンはbobを使用するという明示的指定がなされている。また、明示的に指定されていないクラスには最新バージョンを使用するというdefaultの記述がされている。

また、バージョン指定ファイルには、バージョン管理しないクラス名も指定できる。ただし、JDKが提供するjavaパッケージ中のクラスは、特に記述しなくてもバージョン管理の対象からは外している。クラス名の代りにパッケージ名を指定すると、そのパッケージに含まれるすべてのクラスのバージョン指定を同時に行うことも可能である。クラス名によるバージョンの指定とパッケージ名によるバージョンの指定が同時に存在する場合は、クラス名による指定が優先する。

VCLを用いて機能拡張した仮想マシンの構造は図4に示す通りである。拡張の手段としてはJNI（Java Native Interface）[5]を使用した。まず、既存のJava仮想マシンをJNIを用いてロードする。続いて、VersionClassLoaderを、既存のクラスローダによってJava仮想マシンにロードする。続いて、VersionClassLoaderによりユーザが指定したクラスをロードし、実行する。

ここでひとつ注意しておく点は、既存のクラスローダでなく、VCLによってロードされたクラス内で別のクラスを参照している場合である。Java言語には、java.lang.ClassLoaderのサブクラスとして定義されるクラスローダによってロードされたクラスの内部で参照されるクラスは、同一のクラスローダでロードされるという性質がある。従って、ユーザプログラムのmainメ

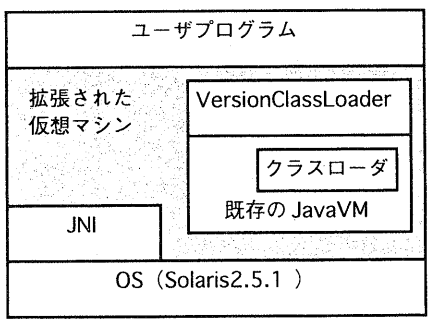

図 4：拡張されたJava仮想マシンの構造

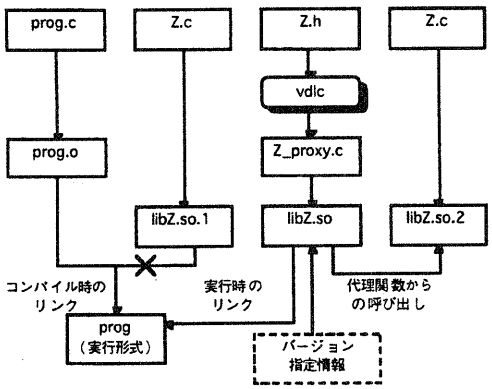

図 5：vdlc を用いた動的発展機構

ソッドを持つクラスのみをVCLでロードしておけば、そのプログラム内で参照される他のクラスもすべてVCLでロードされることになり、バージョン管理を受けられることになる。

このように、Java仮想マシンに VCL をロードすることによって、Java仮想マシンのソースコードには全く変更を加えること無く Java 仮想マシンを拡張し、実行時におけるバージョン管理機能を導入することができた。

4. 間接的動的リンクの実現方式

我々は、Sun の Solaris の動的ライブラリ [4] および、C言語というプラットフォームで、間接的動的リンクを実現した。開発したツールで核となるものは、代理関数の生成を自動的に行うためのコンパイラ vdlc (Versioned Dynamic Library Compiler) である。

vdlc は、ライブラリの関数プロトタイプを含むヘッダファイルを解析して、代理関数を自動的に作成する。例えば、図5に示すように、libZ.so というライブラリのソースがZ.cとZ.hに記述されているとすると、vdlc は、Z.h を読み込んで、Z_proxy.cというファイルを生成する。このファイルには、Z.cに含まれている関数を間接的に呼び出す代理関数が含まれている。そこで、Z_proxy.cをコンパイルしてライブラリを作成する。この時、ライブラリファイルにはバージョン番号を指定しない。Solaris では、ローダは標準でバージョン番号のないライブラリを動的にロードして実行するようになっている。すると、他のバージョンのライブラリとリンクされたユーザプログラムであっても、実行時には自動的に代理関数を含むライブラリをロードし、代理関数を呼び出してくれる。従って、ユーザプログラムには一切変更を加えずに、バージョン管理機能を利用することができるようになる。図5では、当初はライブラリのバージョン1とリンクされたユーザプログラムであっても、実行開始時に代理関数を含むライブラリとリンクされ、実行中には、代理関数を経由してライブラリのバージョン2をアクセスしている所を表している。

Solaris の動的結合ライブラリでは、ユーザプログラムと代理関数の生成する name space と、代理関数によって呼び出されるライブラリ関数の生成する name space が独立している。従って、代理関数と、それが呼び出すライブラリ関数自身が同じ名前であっても、名前の衝突は起こらない。従って、ライブラリ関数も、ユーザプログラムも一切変更せずに、代理関数を生成しさえすれば、バージョン管理が利用できるようになったところに大きな特長がある。

また、現在さらに、C++ 言語においても同様な代理関数（クラス）生成コンパイラの実現を検討中である。

5. 評価と今後の課題

これまで述べてきた通り、ライブラリに複数のバージョンを用意し、それらを、実行時に選択する機能は、C言語およびJava言語というプラットフォームでは、ほぼ実現が完了している。今後の課題として残っていることは、他の言語への対応、および、実行時のロード済みのバージョン

の入れ換え等であるが、これらについてはまだ検討段階で、システムの作成には至っていない。実行時のロード済みのバージョンの入れ換え機能の実現方式としては、derivation [2] を用いて実現していく予定である。

参考文献

[1] Lindholm, T., Yellin, F., The JavaTM Virtual Machine Specification, http://www.javasoft.com/docs/books/vmspec/html/VMSpecTOC.doc.html

[2] Sugiyama, Y. Producing and Managing Software Objects in the Process Programming Environment OPM. in Proceedings of APSEC'94, pages 268-277, Tokyo, Japan, IEEE and IPSJ, 1994.

[3] 杉山安洋、Java クラスの動的バージョン管理の構想、情報処理学会研究報告　97-SE-115、1997 年 7 月

[4] Sun Microsystems, Linker and Libraries Guide

[5] Sun Microsystems, JNI - Java Native Interface, http://www.javasoft.com/products/jdk/1.1/docs/guide/jni/index.html

[6] Sun MicroSystems, RMI Specification, http://java.sun.com/products/JDK/1.1/docs/guide/rmi/index.html