社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE

オブジェクトの動的バージョン管理機能を持つ Java 仮想マシンの試作

倉持 傑† 杉山 安洋‡

† 日本大学大学院工学研究科 〒963-8642 福島県郡山市田村町徳定字中河原 1
‡ 日本大学工学部 〒963-8642 福島県郡山市田村町徳定字中河原 1
E-mail: † kuramoti@ssl.ce.nihon-u.ac.jp ‡ sugiyama@ce.nihon-u.ac.jp

あらまし 実行中のソフトウェアを停止する事なく，機能の追加や変更を可能にするため，我々の研究室では，オブジェクトの動的バージョン管理機能について研究してきた．今回この動的バージョン管理機能を持った Java 仮想マシンを試作した．本稿では，その概要と通常の Java 仮想マシンとの実行速度等の比較を行う．

キーワード バージョン管理，オブジェクト，Java 仮想マシン

A Prototype of the Java Virtual Machine with the Dynamic Version Management Mechanism for Objects

Suguru KURAMOCHI† Yasuhiro SUGIYAMA ‡

† Graduate School of Engineering, Nihon University Koriyama, Fukushima, 963-8642 Japan
‡ Department of Computer Science, College of Engineering, Nihon University Koriyama, Fukushima, 963-8642 Japan
E-mail: † kuramoti@ssl.ce.nihon-u.ac.jp ‡ sugiyama@ce.nihon-u.ac.jp

Abstract We are developing a mechanism that allows software engineers to add and change the functionality of running software systems without stopping their execution. We developed a prototype of the Java virtual machine that includes the dynamic version management mechanism for objects. This paper will give a brief overview and the performance evaluation of the prototype system.

Keyword version management, objects, Java virtual machine

1. はじめに

ソフトウェアの開発作業においては，バグの修正や機能の追加・変更などの理由により，多くのバージョンのソースファイルが作成される．ソフトウェアのリリース後も新しいバージョンをリリースするため開発は継続され，バージョンごとにソースファイルは増え続けることとなる．しかし，多くのバージョンのソースファイルを管理することは容易ではない．そのため，ソースファイルのバージョン管理については古くから研究されており，数多くのバージョン管理用ツールが開発されてきた．その代表として RCS[1]や SCCS[2]などのツールが挙げられる．これらのツールは，ソースファイル単位でバージョン管理を行うものであり，バージョン履歴の保存，ディスクスペースの節約，ソースファイル編集時の排他制御などの機能を持っている．しかし，バージョン管理が必要なものは，ソースファイルのみではない．

近年，ソフトウエアシステムは人間の生活に深く浸透している．それらのシステムの停止は，例え一時的なものであっても，人間生活に大きな影響を与えることが多い．銀行のオンラインシステムや各種のサーバシステムなどがその例である．しかし，ソフトウェアシステムの機能追加や不良の修正のためのバージョンの入れ替え作業は，ソフトウェアを停止させて行われているのが現状である．そのようなバージョンの更新作業はソフトウェアを停止させないで行えることが望ましい．

我々の研究室では，ソフトウェアの実行中に，ソフトウェア中のオブジェクトのバージョンを入れ替えることにより，ソフトウェアの機能変更やバグ修正を行うための，オブジェクトの動的バージョン管理機能[3]について研究してきた．それを Java 言語で書かれたシステムに対して実現するためのシステム[4]が開発されている．

しかし，これまでの研究では，動的バージョン管理システム自身が Java 言語により実装されており，実行効率に問題があった．我々は，現在，この問題を解決するべく，動的バージョン管理の機能を持った Java 仮想マシンの開発を行っている．本論文では，現在までに開発された動的バージョン管理機能を持つ Java 仮想マシンの概要と，その評価について述べる．

2. 動的バージョン管理の概要

Java 言語のようなオブジェクト指向の言語で開発されたソフトウェアの機能を変更するためには，その中で使用されているオブジェクトに変更を加える必要がある．しかも，ソフトウェアを停止させないで機能変更するためには，オブジェクトへの変更を実行中に行う必要がある．我々の研究室ではオブジェクトの動的バージョン管理機能によりこれを実現している．これは使用中のオブジェクト自身に変更を加えるのではなく，オブジェクトに機能の異なる複数のバージョンが存在することを可能とし，それらのオブジェクトを実行中に切り替えて使用することを可能とするメカニズムである．

2.1. 代理オブジェクト

動的バージョン管理システムでは，複数のバージョンを持つオブジェクトが，バージョン管理されていることを意識することなく，一般のオブジェクトと同様にプログラム中で使用することができるように，代理オブジェクトを使用する．図 1 に示すように，ユーザプログラムと使用するオブジェクトとの間に代理オブジェクトを置く．ユーザプログラムは代理オブジェクトをオブジェクト自身であると思ってアクセスする．代理オブジェクトは，ユーザプログラムからの処理依頼を受け取り，バージョン指定情報を元に使用するオブジェクトのバージョンを選択する．そして，指定されたバージョンのオブジェクトに処理を依頼し，その結果をユーザプログラムに返す．代理オブジェクトの機能は，新バージョンの生成とバージョン切り替え機能の 2 つからなる．

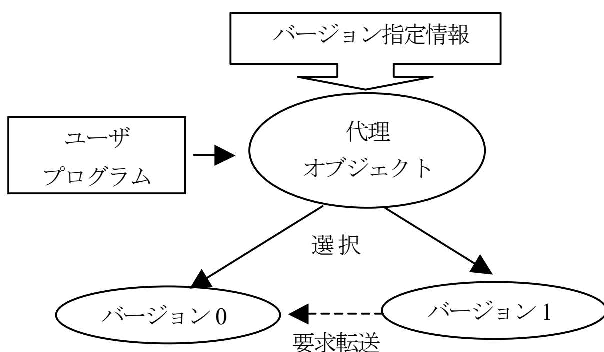

図 1　代理オブジェクト

2.1.1. 新バージョンの生成機能

新バージョンの生成とは，バージョン指定情報により選択されたバージョンが生成されていない場合（例えば，そのバージョンに対する初回アクセス時）にそのバージョンを自動的に生成する機能である．

2.1.2. バージョン切り替え機能

バージョン切り替え機能は，バージョン指定情報を元にアクセスするバージョンを切り替える機能である．その際，指定されたバージョンが存在しない場合は，上記の新バージョンの生成機能を使いそのバージョンを生成した後にそのバージョンへアクセスする．

2.2. バージョンの作成法

また，新しいバージョンを定義する際には，全く新規に定義する必要はなく，古いバージョンからの変更部分を定義するのみで良い．

新しいバージョンに対して，古いバージョンのメソッドの実行要求やインスタンス変数のアクセス要求があった場合には，その要求が古いバージョンに対して転送(delegation)されることにより処理される．

3. 動的バージョン管理の問題点

動的バージョン管理では，上述した通り代理オブジェクトを介しての間接的なアクセスを頻繁に行うこととなり，通常のオブジェクトに対するアクセスに比べて実行効率が悪い．また，新バージョンを生成する作業は決して速いものではない．さらに，新バージョンが再利用している旧バージョンのメソッドを要求された場合，旧バージョンを順に調べそのメソッドが存在する最も新しいバージョンを探す作業なども発生し，この作業も速いものではない．これは，Java 言語にはもともと存在しない delegation の機能を代理オブジェクトの機能として実装しているためである．

代理オブジェクトの動作速度を向上させるためには，上記の一つ一つの処理の速度向上が必要である．しかし，代理オブジェクトの実装を改良しても動作速度を大幅に向上させることはできない可能性が高い．これは，代理オブジェクトの機能の多くが，リフレクションを使用する必要があるためである．そのため，オブジェクトの動的バージョン管理機能自身を Java 仮想マシン[5]に組込むことで，システム全体の実行効率を上げることにした．

4. Java 仮想マシンの概要

Java 仮想マシンには図 2 に示すように実行中に生成されたオブジェクトを格納するヒープ領域が存在する．また，実行時コンスタントプールと呼ばれるデータや，メソッド情報，フィールド情報用の領域も存在する．

Java 仮想マシンの構造は，基本的にはスタックマシンであり，変数の値や一時的な計算結果などをスタックに格納して処理を行う．実行中のスレッドに着目

した場合，そのバイトコードを実行するためにオペランドスタックとローカル変数が存在する．バイトコードの種類により，これらのどの領域に影響を及ぼすかが決まる．

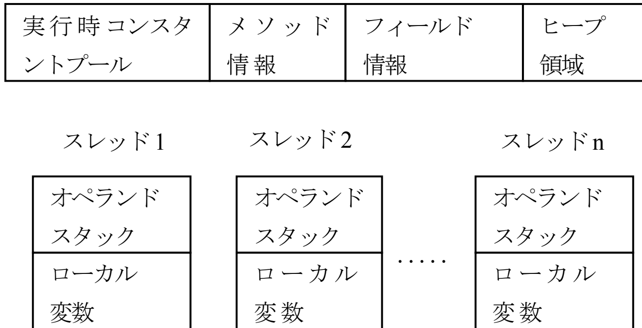

図 2　Java 仮想マシンの構造

4.1. 実行時コンスタントプール

実行時コンスタントプールは，定数やクラス名などを保持するための構造体であり，可変長の配列となっている．参照の際には，インデックスを指定する．実際のエントリは図 3 のようになっている．

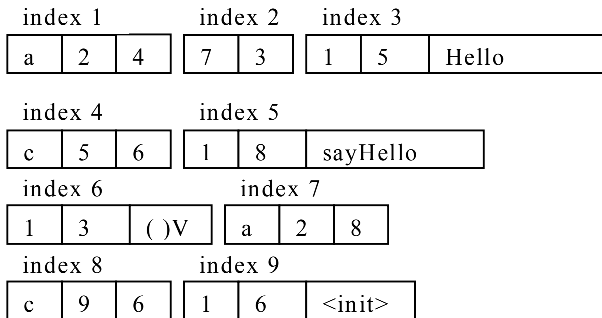

図 3　コンスタントプール

エントリの最初の 1 バイトは常にタグとなっており,そのエントリが示す情報を識別するために用いる．このタグの種類の例を表 1 に示す．

表 1　コンスタントプールのタグ

定数の型	値	意味
CONSTANT_Class	7	クラス
CONSTANT_Methodref	a	メソッド
CONSTANT_NameAndType	c	名前，引数 戻り値
CONSTANT_Utf8	1	文字列

図 3 のインデックス 1 のエントリは，タグが a となっていることからメソッドに関する情報を示すことがわかり，その中でさらに，インデックス 2 と 4 を参照するようになっている．インデックス 2 のエントリはタグ 7 でクラスを示しており，さらにインデックス 3 のエントリを参照するようになっている．インデックス 3 のエントリは，タグ 1 で文字列が格納されていることを示している．インデックス 4 のエントリは名前と型を示しており，その中でインデックス 5 と 6 を参照するようになっている．インデックス 5 と 6 は，ともに文字列が格納されている．インデックス 6 のエントリは引数，戻り値が空であることを表している．

このようにたどっていくと，インデックス 1 のエントリは Hello クラスのメソッド void sayHello()を示しているということになる．

同様にして，インデックス 7 のエントリは Hello クラスの名前が<init>，引数と戻り値が void のメソッドを示しているが，インデックス 9 の示す<init>は特殊な表記でありコンストラクタを示すものである．つまり，Hello()というコンストラクタをインデックス 7 は示しているということになる．

コンスタントプールは，もともとクラスファイル中に存在し，クラスをロードした時点で実行時コンスタントプールとして仮想マシン上にロードされる．

4.2. 仮想マシンの実行例

下記のソースコードの一部を例とし，実行例を説明する．

```
hello = new Hello( );
hello.sayHello( );
```

このソースコードをコンパイルすると，次のようなバイトコードが生成される．

```
1:new #2 <Class Hello >
2:dup
3:invokespecial #7 <Method Hello( ) >
4:putfield #10 < Field Hello hello >
5:getfield #10 < Field Hello hello >
6:invokevirtual #1 <Method void sayHello()>
```

これはコンパイラにより生成されたバイトコードを，javap コマンドで見やすい形で表示したものである．Javap コマンドは JDK に含まれるコマンドでクラスファイルに含まれる情報を表示することができるツールである．オペコードの後の#のついた数字は実行時コンスタントプールへのインデックスであり，そのオペコードの引数である．その後の< >の部分はそのインデックスが指す実行時コンスタントプールの情報であり，javap のつけたコメントである．

バイトコードとヒープ領域，オペランドスタックの関係を図 4 を用いて説明する．

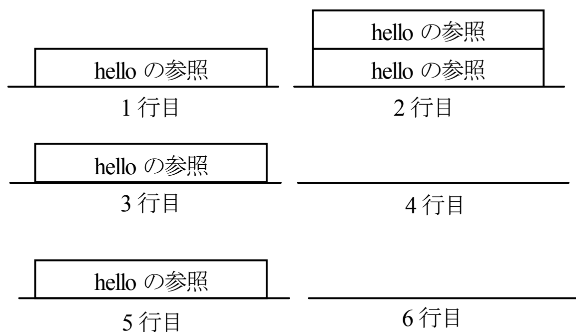

図 4　スタックの状態

1 行目の new では，Hello クラスのインスタンスをヒープ領域に生成し，その参照をオペランドスタックに push している．

2 行目の dup では，1 行目でオペランドスタックに格納された値をコピーし，オペランドスタックに push する．つまり，オペランドスタックに同じ参照が 2 つ存在する状態となる．

3 行目の invokespecial では，Hello クラスのコンストラクタを呼び出している．この際，オペランドスタックの値を pop する．この場合はオペランドスタック中の Hello クラスのインスタンスの参照を pop することとなる．

4 行目は変数 hello にオペランドスタックから pop した値を格納する．この場合はオペランドスタック中の Hello クラスのインスタンスへの参照が変数 hello に格納されることとなる．

5 行目では，変数 hello の値をオペランドスタックに push している．変数 hello には Hello クラスのインスタンスへの参照が格納されているので，それがオペランドスタックへ push される．

6 行目では，オペランドスタックから Hello クラスのインスタンスの参照を pop し，そのオブジェクトの sayHello というメソッドを実行している．

5. 仮想マシンでの代理オブジェクト機能の実現

動的バージョン管理機能を組み込んだ仮想マシン JVM/DVM (Java Virtual Machine / Dynamic Version Management)は，図 5 のように既存システムでの代理オブジェクト相当の機能を持たせることで実現する．

今回の開発に用いている Java 仮想マシンは Kaffe [6] である．これは，オープンソースの Java 仮想マシンである．オープンソースなので情報が多く，開発を比較的容易に行えると考えたためこの仮想マシンを採用した．Kaffe の仮想マシンにはインタプリタ版と JIT 版がある．今回は，動作速度は JIT に比べて遅いがソースコードを理解しやすいインタプリタを選択した．Kaffe を改造した仮想マシンを kaffe/DVM と呼ぶ．

代理オブジェクト相当機能を Kaffe/DVM に組み込むためには，新バージョンの生成機能と，バージョン切り替え機能を実装する必要があるが，今回の試作では，バージョン切り替え機能のみを実装した．プログラム中で使用するオブジェクトのバージョンを実行時に生成することはできないが，それらのバージョンを予め生成しておき，それらを実行時に切り替えて使用することができる．なお，バージョンの生成機能は，仮想マシンの外部で，Java 言語のソースコードレベルで行っている．

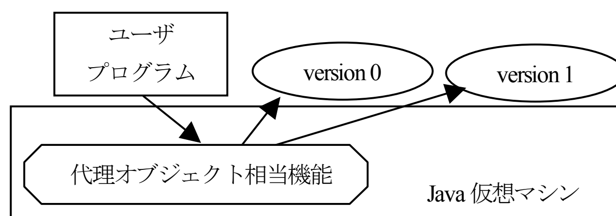

図 5　JVM / DVM

仮想マシン内部でバージョン切り替え機能を実現するためには，アクセス先オブジェクトを動的に変更する必要がある．メソッド呼び出しのバイトコードである invokevirtual が実行される際には，4.2 節で述べた通り，オペランドスタックよりオブジェクトの参照を得る．4.2 節の例では，getfield によってインスタンス変数中に格納されたオブジェクトの参照を得ている．

そのため，getfield を実行する際，その変数がバージョン管理の対象であるか否かを判定し，対象であった場合には，現在使用するべきバージョンのオブジェクトの参照をオペランドスタックに push する事で，invokevirtual によるメソッド呼び出しが切り替えられる事となり，バージョン切り替え機能を実現する事ができる．

なお，getfield 以外にも aload，getstatic 等オペランドスタックにオブジェクトの参照を格納する命令は存在するため，それぞれのバイトコードで getfield と同様の処理を行う必要がある．

6. Kaffe/DVM の試作

6.1. kaffe の構成

仮想マシン kaffe は，kaffe，kaffevm，kaffevm/intrp 等のモジュールから構成されている．モジュール kaffe は，kaffe コマンドに直結する処理を行うもので，オプションの判別や，Kaffe のバージョン情報の出力などを担当する．この中の main 関数で仮想マシンを実際に開始するモジュールを呼び出している．

モジュール kaffevm/intrp は，Java 仮想マシンの実装であり intrp は，インタプリタを表している．今回は，インタプリタを選択したのでこれを使うが，他に kaffe/jit が存在する．kaffevm/intrp の中の，machine 関数によって実際の仮想マシンの動作を行う．上記の main 関数によって，この関数 machine が呼ばれる．関数 machine は，基本的にはループしてバイトコードをひとつずつ実行する．そのループの中で，バイトコードごとに分岐し，それぞれを実行する形になっている．

kaffevm の中の kaffe.def がそのバイトコードごとの処理を担当する．つまり，kaffe.def には，バイトコードの処理がすべて定義されているのである．

6.2. kaffe への変更

今回の実装では，バージョン管理されたオブジェクトの各バージョンの参照は，それぞれ別のインスタンス変数へ格納する方式とした．getfield の引数は，その変数の名前やクラス名等の情報にリンクされたインデックスである．このインデックスを必要なバージョンを格納する変数を表すインデックスと交換することで，オペランドスタックには切り替え後のバージョンの参照が格納されることとなる．

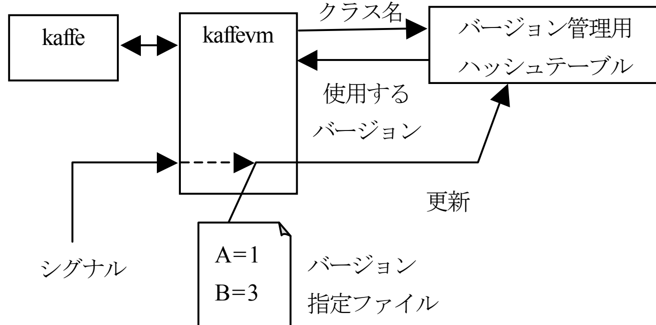

図 6 Kaffe / DVM の構成

そのインデックスの切り替えを実現するために，ハッシュテーブルを用意した．このハッシュテーブルは，クラス名をキーとし，使用すべきバージョンのインデックス情報を得られるものである．getfield が実行されると，引数からたどりクラス名を得る．そして，そのクラス名を元にハッシュテーブルを検索し，見つかれば交換先のインデックス情報を得て，そのインデックスを getfield の引数として実行する．

今回試作の仮想マシンにはオプション指定することで，バージョン管理機能の有効，無効を切り替えられるようにした．これにより，デフォルトでは getfield を行う度にハッシュを検索する作業などは発生しない．

ハッシュテーブルの生成は，バージョン指定情報ファイルを元に行う．この指定情報ファイルは，例えば A クラスのバージョン 1 で稼働させる場合は A = 1 のように記述したテキストファイルである．

ハッシュテーブル更新のタイミングは，バージョン指定情報ファイルを変更した後にユーザにより仮想マシンに対してシグナルを送る形とした．このシグナルが送られるまでは，ハッシュテーブルの更新が行われないため，バージョンも切り替わることはない．シグナルが送られる以外に，仮想マシンの起動時にもハッシュテーブルの更新は行われる．

上述のように，バージョン指定情報ファイルを元にハッシュテーブルを作成し，そのハッシュテーブルを用いて，getfield の引数を変更することが可能となった．これによって，インスタンス変数に格納されたバージョン管理対象のオブジェクトのバージョンを切り替えて使用できるようになった．

6.3. 比較検証用サンプル

今回開発した Kaffe/DVM の実行効率を検証するため，次のようなサンプルプログラムを用意した．基本構成は，A クラスをバージョン管理対象とし，B クラスの中で A クラスのインスタンスを生成し，そのメソッドを 100 回，150 回，200 回呼び出すというものである．対象のメソッドは，System.out.println による出力， Math.sqrt，Hashtable の操作の 3 種類を用意した．

System.out.println, Math.sqrt を実行するメソッドは，それぞれ 1 回該当の処理を行っている．Hashtable の操作をするメソッドについては，Hashtable に 100 項目データを追加し，その追加した 100 項目を参照するという処理を行う．

3 種類のバージョン管理対象のオブジェクトのメソッドを，通常の Kaffe 仮想マシン，今回開発した Kaffe/DVM，代理オブジェクトを用いた動的バージョン管理システム DVMS でそれぞれ実行して計測した．なお，DVMS ではメソッドの初回起動時にバージョン生成機能が働くため，全ての計測は初回のメソッド呼び出しを除外した．

6.4. 比較検証結果

表 2 に，3 種類のサンプルについて計測した結果を示す．これは，各メソッドを 100 回，150 回，200 回繰り返して実行する時の処理時間を 10 回計測し，その平均を求めたものである．

System.out.println や Math.sqrt については，以前のシステムと比較して今回試作したシステムが圧倒的に高速であると言える．

しかし，Hashtable に関しては以前のシステムの方

が速いという結果が得られた．これは，システムの試作構想段階では，予想していなかった結果である．その原因として，getfield の対象となるオブジェクトがバージョン管理対象であるか否かの判定をするため，getfield 命令が実行される度に仮想マシン内に組込んだバージョン管理情報のハッシュテーブルに問い合わせを行うためであることが考えられた．getfield の対象となるオブジェクトがバージョン管理の対象となっていない場合は，以前のシステムではその処理に伴うオーバヘッドは発生しない．従って，バージョン管理に関係しない getfield の実行回数が多くなるにつれて，その処理時間が問題となる．

表 2　平均処理時間

サンプルプログラム	繰り返し回数	Kaffe	Kaffe/DVM	DVMS
println	100	145.8	165.9	770.3
	150	219.7	247.3	1139.8
	200	296.7	324.8	1520.8
sqrt	100	1.3	1.4	553
	150	2	2.1	827.4
	200	2.5	2.9	1115.7
Hashtable	100	3636.6	4431.9	4245.0
	150	5424.8	6630.4	6348.6
	200	7265.8	8852.6	8640.1

(単位：ms)

一方 Hashtable 以外のサンプルについては，期待した通りの結果が得られているが，これについては，バージョン管理されていないオブジェクトを対象とする getfield の実行回数が少ないため，以前のシステムでのリフレクションによるオーバーヘッドよりも getfield のオーバヘッドが少ないためであると考えられる．

表 3　getfield の実行回数とオーバヘッド

サンプルプログラム	getfield の実行回数 (A)	オーバヘッド (B)	getfield 一回あたりのオーバヘッド (B)/(A)
println	11000	158.9ms	0.014ms
sqrt	100	1.5ms	0.015ms
Hashtable	262200	4420.7ms	0.016ms

この推測を検証するため，表 3 に各サンプルでのメソッド呼び出し 100 回あたりの getfield の実行回数(A)と，その際に発生する Kaffe/DVM におけるオーバヘッド(B)を示す．(B)は表 2 における 200 回と 100 回の実行時間の差を求めたものである．getfield の実行回数は，Hashtable のサンプル実行時に突出して多い事が確認できた．これにより，上述した通り getfield の回数が多いため旧システムでのオーバーヘッドより，今回のシステムの方が遅くなったと言える．さらに，getfiled を一回処理するための Kaffe/DVM のオーバヘッドは，サンプルによらずほぼ一定であることも確認できた．

7. まとめと今後の課題

オブジェクトの動的バージョン管理システムの実行効率を改善するため，バージョン切り替え機能を組み込んだ Java 仮想マシン Kaffe/DVM を試作し，そのメリット，デメリットを確認する事ができた．

代理オブジェクトを用いたシステムと比較し，今回試作したシステムが必ずしも優れている訳ではない．今回の試作したシステムを有効に使用できるのはバージョン管理の対象とならないオブジェクトを対象とする getfield が少ない時である．

今後の課題としては，バージョン切り替え機能の改善と，今回は実装できなかった新バージョンの動的生成機能の組み込みが挙げられる．

バージョン切り替え機能については，今回の評価の対象としたのは getfield のみであった．しかし，getstatic，aload 等におけるバージョン管理のためのオーバヘッドも考慮すると，実行速度が現状より遅くなる事は確実である．そのため，より効率の良い方式を再度検討する必要がある．

新バージョンの生成機能は，生成の方式や生成タイミング等を検討しなければならない．

また，ユーザがシステムを運用する際，システム中で使用されている各オブジェクトのバージョンの状態を把握できる仕組みも導入する予定である．これにより，バージョン変更作業がより容易になると考えている．

文　献

[1] RCS, http://www.cs.purdue.edu/homes/trinkle/RCS/
[2] Rochkind, The Source Code Control System, TOSE, IEEE, SE-1, No.4, pp.364-370, 1975
[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,pp.49-56,1998.
[4] 杉山安洋，戸部勝文，動的バージョン管理に基づく実行中のソフトウェア発展法，情報処理学会研究報告，98-SE-119,pp.49-56,1998.
[5] ティム・リンドホルム，フランク・イェリン，Java 仮想マシン仕様　第 2 版，株式会社ピアソン・エデュケーション，2001
[6] Transvirtual Technologies, Kaffe, http://kaffe.org