ソフトウェア工学 115－7
（1997．7．31）

Javaクラスの動的バージョン管理の構想

杉山 安洋

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

ソフトウェアは、その開発作業が進行するにつれて、数多くのバージョンが作られる。一度完成してユーザにリリースした後でも、不良の修正などで新しいバージョンが継続的に開発される。バージョン管理はソフトウェア工学においては古くからある研究分野であり、これまでも、SCCS やRCS に代表される優れたバージョン管理ツールや、DSEE 、Adele などのバージョン管理機能を持ったシステムや開発環境が数多く開発されてきた。しかし、これまでのC++のような言語で開発されたソフトウェアと、Java言語で開発されたソフトウェアでは、バージョン管理で必要となる要素が大きく異なってくる。本稿では、Java におけるソフトウェア開発には、動的なバージョン管理機能が必要であることを述べ、さらにそれを実現する方式についての構想を述べる。

Dynamic Version Management of Java Classes

Yasuhiro Sugiyama

Department of Computer Science
Nihon University
Koriyama, Japan
sugiyama@ce.nihon-u.ac.jp

Version management is a research area that has a quite long history. Quite a few useful version control tools, including SCCS and RCS, have been developed. Software development environments, like DSEE and Adele, which include a versions management mechanism also have been developed. However, versions of Java programs need to be produced, managed, and selected quite differently from traditional programming languages, like C++. Particularly versions in Java need to be managed and selected dynamically at the execution time, in addition to the development time. This paper presents the requirements for dynamic version management in Java, and our approach to implement the requirements.

1．はじめに

ソフトウェアは、その開発作業が進行するにつれて、数多くのバージョンが作られる。一度完成してユーザにリリースした後でも、不良の修正などで新しいバージョンが継続的に開発される。バージョン管理はソフトウェア工学においては古くからある研究分野であり、これまでも、SCCS [10]やRCS [18] に代表される優れたバージョン管理ツールや、DSEE [9] 、SVCE [6]、Adele [4] 、Gypsy [2]、CEDAR [8] などのバージョン管理機能を持ったシステムや開発環境が数多く開発されてきた。しかし、これまでの言語で開発されたソフトウェアと、Java言語 [16] で開発されたソフトウェアでは、バージョン管理で必要となってくる要素が大きく異なってくる。本稿では、Javaにおけるバージョン管理において必要がある事項をまとめ、それらを実現する方式についての構想を述べる。

本稿の構成は次の通りである。まず、第2節で、これまでのバージョン管理ツールの主な機能をまとめる。次に第3節で、Java言語 によって記述されたソフトウェアがこれまでの言語で記述されたソフトウェアと異なる点を簡単にまとめる。続いて第4節で、Java言語でソフトウェアを開発する時に必要となるバージョン管理とはどのようなものかを明らかにする。第5節では関連分野の研究をまとめ、第6節で実現方針を述べ本稿のまとめとする。

2．これまでのバージョン管理

これまでのSCCSやRCSなどに代表されるバージョン管理ツールやシステムは、ソースファイル単位でバージョンを管理し、ソフトウェアの開発中やメンテナンス中にソフトウェア技術者が対処しなければならない問題を支援してきた。これまでのソースファイルのバージョン管理には、いくつかの重要なポイントがあった。

まず、第一点は、ソースファイルの変更履歴の保存と管理である。ソフトウェアの開発中には、ソースファイルを継続して修正する。その際、行う修正がいつも正しいとは限らず、場合によっては変更を取り消す必要が出てくる。その際に必要なのが変更履歴の保存である。そうすることにより、必要に応じて変更前のバージョンに戻ることができる。

第二点は、ディスクスペースの有効利用である。関連したバージョンは、似ていることが多く、それらをそのまま保存しておいたのでは、ディスクスペースを有効に活用することができない。そこで、複数のバージョン間の差分（デルタ）のみを保存することにより、似たバージョンの保存の際に発生するディスクスペースの無駄を最小限にとどめてきた。

第三点は、複数のプログラマによる作業の同期である。複数のプログラマによる同一ファイルの同時変更による衝突をさけるため、チェックイン、チェックアウトなどのコマンドにより、バージョンの変更や作成の同期をとる機能を提供していた。

3．Java言語の特徴

Java言語の登場は、コンピュータソフトウェアの実行方式に大きな変革をもたらしている。これまで、コンピュータプログラムは、各コンピュータの持つハードディスク等の記憶媒体からメモリにロードされ、CPUに実行されるのが通例であった。

Javaの特徴の第一点は、ネットワークで結合された遠隔地のサーバのハードディスクに格納されたプログラムをネットワークを通してダウンロードし、それをクライアントで実行することが可能となったことである。Javaでは、ソースファイルをコンパイルすると、CPUのアーキテクチャには依存しないバイトコードが生成される。それが、サーバからクライアントにダウンロードされると、そのクライアントのJava仮想マシンで実行される。従って、単一のバイトコードを複数の異なるアーキテクチャのクライアント計算機上で実行することができる。もちろん、これまでも、NFS [11] などのネットワークファイシステムを使えば、離れた計算機のハードディスク上のソフトウェアを実行することはできた。しかし、NFSはあくまでファイルシステム自身を共有することが目的であり、Javaのようにファ

イルシステムを共有しないサーバとクライアント間でプログラムをダウンロードして実行する方式とは大きく異なるものである。しかも、これまでの言語で記述されたソフトウェアは、コンパイル、リンクされて、各CPU特有の実行形式ファイルへと変換され、それらのCPUに直接実行されていた。従って、Javaのように、ひとつのソフトウェアを複数の異なったアーキテクチャの計算機上で実行することは、不可能であった。

さらに、これまでは、プログラムを一括してメモリ中にロードして実行するのが通例であった。しかし、Javaでは、プログラムがクラスという機能単位に分割されており、ダウンロードする際にもプログラム全体をダウンロードするわけではなく、必要なクラスを必要な時にダウンロードして実行するのが、その標準となっている。もちろん、これまでも、オーバレイのように、一括してメモリにロードせずに、分割して必要な部分のみをメモリにロードして実行する方式もあった。これはあくまでもメインメモリの容量の不足を補うためなどの別の目的のためであり、分割してロードすること自体が、その目的ではなかった。

4．Javaのバージョン管理の必要事項

Java言語の登場により、これまでのソースファイル中心のバージョン管理は大きく変化する必要がでてきている。Java言語においても、プログラムはソースコードとしてファイル中に記述される。従って、ソースファイルレベルでのバージョン管理機能はもちろん必要であるが、それだけでは十分ではない。Javaにおけるバージョン管理は、次に示す3点に注意して実現する必要がある。

4．1．実行時のバージョン管理

Java言語におけるバージョン管理において必要な点の第一は、実行時のバージョン管理である。これまで、バージョン管理というのは、基本的には、ソフトウェア開発時の作業であった。これは、ソースファイル単位でバージョンを管理しておき、適切なバージョンのソースコードをコンパイル、リンクして実行形式ファイルを作

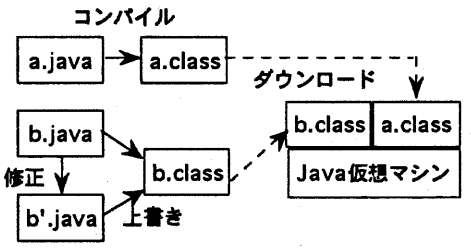

図1：現在のJavaにおけるプログラムの修正

成しておけば、その後のソースコードの変更から、実行形式ファイルは隔離されていたからである。即ち、実行時には、ソースコードのバージョンを心配する必要はなかったし、また、実行時にバージョンの管理はできなかった。しかし、Javaには、この原則は当てはまらない。

Java言語で記述されたプログラムは、コンパイルされるとクラスごとのバイトコードに変更されることは前に述べた。これらのクラスのバイトコードは、一つの大きな実行形式ファイルにリンクされるわけではない。サーバ側に、各クラスごとにバイトコードのまま格納され、クライアントからの要求に従ってダウンロードされ使用される。従って、図1に示すように、一度コンパイル済みのクラスであっても、変更を加えてそれを再度コンパイルしておけば、そのバージョンが実行時にも使用されることになる。

これには、良い面も不便な面もある。利点のひとつに、プログラムを実行時に発展させることができることがあげられる。実行に必要なクラスは、必要になった時にダウンロードされる。従って、クラスがダウンロードされるまでは、プログラム中の他のクラスとの整合性を失わない範囲で、自由に変更を加えることが可能となる。また、これまで、ソフトウェアの不良が修正されたり、機能が追加されたりして、ソフトウェアの新しいバージョンが登場すると、各ユーザが手持ちのバージョンを最新のバージョンに入れ換えたりする作業は、すべてユーザの責任で行われていた。しかし、コンピュータにあまり詳しくない一般のユーザにとっては、このバージョンの入れ換えという作業は、容易なもので

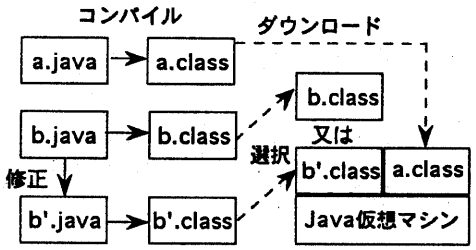

図２：バージョン管理機能に基づく Javaプログラムの修正

はない。しかし、このJavaの機能を用いれば、サーバ側で常に最新のバージョンのソフトウェアを用意しておきさえすれば、一般のユーザがバージョンの更新などの作業を行わなくても、常に最新のバージョンのソフトウェアが使用できるという大きな利点がある。

一方で、これは、ひとつのクラスが複数のアプリケーションで共有して使用される場合などに問題を引き起こす場合がある。また、クラスが他のクラスを参照している場合、一方のクラスに変更を加えてしまうと、他方のクラスとの整合性が失われてしまう場合も出てくる。例えば、二つのプログラムＡとＢが、共通のクラスＣを使用するとする。プログラムＡの開発者がクラスＣのあるバージョンを自分のプログラムで使用しようとして、バイトコードにコンパイルしておいたとしても、プログラムＢの開発者が、そのクラスを変更して再度コンパイルしてしまえば、プログラムＡとＢのどちらのユーザも、プログラムＢの開発者が変更したクラスを用いて、プログラムＡとＢを実行することになる。これは、バイトコードレベルでのバージョンの管理の必要性を表しているだけではなく、複数のクラス間でのバージョンの整合性の管理を、実行の際に行う必要があることを示している。

また、これは、見方を変えれば、図２に示すように、サーバが一つのクラスの複数のバージョンのバイトコードを用意しておけば、ユーザが実行時に選択して実行できることを示している。従って、バージョン管理はソフトウェア開発中だけに必要となる作業ではなく、ソフトウェアの実行中にソフトウェアのユーザまでもが気を配る必要がある（あるいは気を配ることができる）問題となったのである。もちろん、すべてのユーザがこの機能を使用するわけではない。コンピュータにあまり詳しくない一般のユーザには、何も指定しなくいても、適切なバージョンが使用できるようなメカニズムを提供する必要がある。

４．２．クラス単位のバージョン管理

Javaにおけるバージョン管理がこれまでのバージョン管理と異なる第２の点は、バージョンを管理する対象と、その粒度の変化である。これまでは、ソースファイルをコンパイルしてオブジェクトコードを作成し、それらをリンクして実行形式ファイルを作成するという一連の作業は、すべてファイルを単位に行われていた。そのため、これまでのバージョン管理ツールは、ソースファイル単位でバージョンを管理すれば十分であり、そのような機能のみを提供してきた。しかし、Javaではこの原則は通用しない。もちろん、ファイルシステム以外のソフトウェアデータベースを持つ開発環境においては、ファイルではなく、データベースのオブジェクト単位でバージョンは管理されるわけであるが、ここでは、ファイルシステムベースの開発環境を考える。

Java言語においては、多くのオブジェクト指向言語と同様に、プログラムをクラスに分割して開発する。従って、開発作業中には、クラスごとに数多くのバージョンが作られることになり、これらのクラス毎のバージョン管理が必要となってくる。C++などの言語では、ひとつのソースファイルに複数の関数やクラスを定義しておいたとしても、それをコンパイルすればひとつのオブジェクトファイルが生成された。しかし、Javaでは、ひとつのソースファイルに複数のクラスを記述した場合、それをコンパイルすると、クラスごとのバイトコードファイルが生成される。従って、これまでのように、ソースファイルとオブジェクトファイルが一対一の関係を持っ

ているわけではない。しかも、前述したように、バイトコードのサーバからクライアントへのダウンロードはクラス単位で行われる。従って、ソースファイルごとではなく、クラスごとおよびバイトコードごとのバージョン管理が必要となる。

クラスごとのバージョンを管理するためには、同一クラスの複数のバージョンを識別できる必要がある。クラスも基本的にはソースファイルに記述されるので、これまでのように、ソースファイルの拡張子を用いて、そのバージョンを区別することは可能である。ひとつのファイルにひとつのクラスのひとつのバージョンを定義しておけば、ソースファイルレベルで、複数のバージョンを混同することはなかろう。しかし、一度それらをコンパイルしてしまうと、現行のJavaコンパイラでは、図1で見た通り、すべて同名のバイトコードファイルが生成されてしまい、識別が不可能となる。従って、これまでのように、ソースファイルレベルでバージョンが識別できれば十分というわけではなく、バイトコートレベルで識別できる必要がある。

さらに、Javaプログラムはアプレットという形で、HTML文書から呼び出されて実行される場合もある。従って、Java言語で記述されたクラスのバージョンのみではなく、HTML文書との整合性を確認する機能も必要となってくる。もちろん、現在のHTML文書には、バージョンを管理する機能が含まれていないので、HTML文書のバージョン管理の機能も必要となってくる。

4．3．実行時のビルドプロセス管理

バージョン管理と並んで話題に上がるのがビルドプロセス管理である。ビルドプロセスを実行する際には、そこで使用するソースコードのバージョンを選択する必要があるため、ビルドプロセス管理ツールはバージョン管理ツールと密に連携していることが多い。Make [5] に代表されるビルドプロセス管理ツールは、実行形式ファイルを作成する作業をスクリプトにより定型化すると同時に、ソースファイルなどの変更を行った際にコンパイルやリンク作業を忘れるなどの単純な作業ミスを防ぎ、実行形式ファイルが矛盾なく生成されることを保証してくれる。しかし、Javaの環境下では、ビルドプロセスもこれまでとは異なってくる。

これまでのC++などの言語では、すべてのクラスのコンパイルが完了し、それらをリンクして実行形式ファイルを作成するという作業を行ってから、始めてそのプログラムが実行できた。しかし、Javaでは、すべてのクラスのバイトコードがプログラムの実行を開始する時点で必要であるわけではないので、ビルド作業も実行と並列して行うことが可能となっている。即ち、あるプログラムを実行するためにクラスAを必要とする場合、Aのバイトコードをダウンロードする直前に、ソースコードをコンパイルして生成してからダウンロードするといったことが可能となってくる。

もちろん、実行時にコンパイル作業を行っていたのでは、実行速度が低下することは目に見えて明らかである。しかし、現在のJavaには、JIT(Just In Time)コンパイルのような技術もある。従って、バイトコードの生成は、実行開始前に完了させておくにしても、JITコンパイルは、実行時に行える。即ち、同一クラスの複数のバージョンのダウンロードが完了していれば、JITコンパイルを行う段階で、まだバージョンを選択する機会が残されていることになる。もちろん、対象となるクラスの複数のバージョンを前もってダウンロードしておくことは、ネットワークのトラフィックを増大させるので、スケジューリングやキャッシュの有効利用が必須ではある。

さらに、これまでは、Makeなどのツール内で閉じていたビルドプロセス管理も、このように実行時に動的に行うようになると、Javaの仮想マシン（インタプリタ）との密な連携が必要となる。従って、Netscapeなどのブラウザなどのアプレットを使用する一般のツールとも連携が必要となってくる。

5．関連研究

5．1．オブジェクト指向データベース

オブジェクト指向データベースの分野でおい

ても、オブジェクトのバージョンや、それを定義するクラス（スキーマ）のバージョンの研究が行われている。オブジェクト指向データベースでは、クラスから永続的なインスタンスが生成される。何らかの理由で、クラスを変更する必要が発生した場合などには、すでにインスタンスを持ったクラスを変更しなければならない。このような、永続的なインスタンスを持ったクラスの定義の変更を矛盾なく行うためのメカニズムとして、バージョン管理が注目されている。バージョン管理機能を持ったデータベースの例には、ORION [7]、DAMOKLES [3] などがある。いくつかの研究の比較は [1] にも見られる。本稿では、Zdonik [19] のアプローチを例として検討する。

インスタンスを持つクラスを変更してしまうと、もちろん、そのクラスの定義と生成済みのインスタンスとの整合性が失われてしまう。Zdonik は、クラスは変更不能 (immutable) であると仮定するばかりでなく、クラスに複数のバージョンを持つことを許し、クラスの変更は新たなバージョンの生成を意味すると定義した。さらに、ひとつのバージョンのインスタンスは、そのバージョンが変更されて新しいバージョンが生成されても、いつも同じバージョンのインスタンスであり続ける。これにより、インスタンスを持つバージョンを変更しても、インスタンスとクラスには、整合性が失われないようにした。

この方式では、オブジェクトは、常に同じバージョンのインスタンスである続けるので、クラスとインスタンスの整合性が失われるという問題は解決される。しかし、クラスの変更が、その不良を修正するものであった場合などは、その不良を持ったクラスのインスタンスも一斉に修正する必要がある。その場合には、明示的なインスタンスの属するクラスの変換（Migration）が必要となり、それは複雑な作業となってしまう。

また、ユーザから見ると、同じクラスのインスタンスでも、そのバージョンが異なれば、それを扱うためのインタフェースが異なってしまうという問題が発生してしまう。Zdonikは、この問題には、バージョンセット(Version Set)という概念を導入して問題解決を図った。バージョンセットは、すべてのバージョンのインタフェースの合併である。従って、すべてのバージョンの、プロパティやオペレーションは、必ず、このバージョンセットのプロパティやオペレーションに含まれている。新しいバージョンを作成する際は、このバージョンセットに含まれるプロパティやオペレーションのみを使用するようにするとともに、そのバージョンで定義しないものに対しては、必ず例外処理のためのハンドラを定義することにより、バージョンが変わっても、実行中にエラーでプログラムが停止することが無いことを保証しようとした。しかし、これもバージョンセットという決められた枠内でのクラスの変更しか許さない方式であり、Javaのクラスの変更のように、当初から予想できない変更には対応できないという問題がある。

5．2．動的結合ライブラリ

現在のUNIXなどのリンカには、動的結合機能をもつものが多い。これは、すべてのオブジェクトプログラムを静的にリンクして実行形式のファイルを作るのではなく、プログラムの実行開始時に、Shared Object [17] と呼ばれる形式のオブジェクトライブラリをリンクしてから実行する方式である。

この方式でも、Javaにおけるバイトコードの動的なダウンロードと同様に、静的にリンクされてしまうオブジェクトファイル以外は、実行時まで自由に変更することができるという利点が得られる。これは、OSに付属の基本ライブラリを提供する方式として重要であり、広く用いられている。ライブラリに不良が発見されたり、機能が拡張された場合には、そのライブラリのみを変更するのみで、それを使用するすべてのアプリケーションには変更を加えることなく、新しいライブラリが使用可能となる。

動的結合ライブラリにも、バージョン管理の機能がある。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、リンカが読み出すディレクトリを指定することで、バージョンの明示的な指定も可能とはなる。

しかし、これらは、あくまでリンカの機能であるので、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。

6．まとめと実現方針

本論文では、Java言語を用いて記述されたソフトウェアを開発、実行する際に必要なバージョン管理の機能を明確にしてきた。我々の目標は、通常のソースコードレベルでのJavaクラスのバージョン管理に加えて、次の機能をバージョン管理で実現することである。

1) 実行時のバージョン管理
2) クラス単位のバージョン管理
3) 実行時のビルドプロセスと連携したバージョン管理

実行時のバージョン管理については、まず、Java仮想マシンを変更して、実行開始時にダウンロードするクラスのバージョンを指定できるようにする予定である。しかし、我々の目的は、それだけに止まらず、実行中にバージョンの指定あるいは、使用するバージョンの変更ができることを目標とする。

クラスのダウンロードを完了し、実行を開始したあとでも、そのクラスのインスタンスのデータ部に変更を加えないもので、かつ、他のクラスとの整合性を保存する範囲ならば、メソッド部分の変更をすることは十分可能である。従って、実行時に何か不都合が発生した場合でも、そのプログラムの実行を停止させることなく、プログラムの修正がある程度は可能となる。ただし、その修正がクラスのデータ部の変更を含む場合は、実行を開始してしまったのクラスのバージョンの変更は容易ではない。筆者の研究室では、ソフトウェアデータベース中のオブジェクトの発展や進化を実現するメカニズムとしてデリベーション [12] を考案し、それをソフトウェアプロセスの動的変更 [14] を行うメカニズムに応用してきた。今回は、このデリベーションをJavaに組み込む予定である。

クラス単位のバージョンの識別に関しては、現在2つの方式を検討中である。第一の方式は、クラスを別々のソースファイルに記述してファイル名によってそれらのバージョンを識別する方式である。もちろん、バイトコードも複数のバージョンが持てるようにする。第2の方式は、Java言語の言語仕様を拡張し、クラス自身にバージョンの指定ができるようにする方式である。いずれの場合でも、既存のソフトウェアを変更なくバージョンの追加ができるようにする予定である。

ビルドプロセスとの連携に関しては、筆者の研究室でこれまで開発してきたツールにDoMake(Distributed Object Make) [13,15]がある。DoMakeは、ネットワーク上に分散したソースコードから実行形式ファイルを作成するビルドプロセスを管理するツールである。各ソースファイルのコンパイル作業などは、それが存在するサーバ計算機上で行うことができ、それらを組み合わせる作業がユーザ即ちクライアントの計算機上で行える。従って、ビルドプロセスを複数の計算機上に分割することが容易に行える。DoMakeは、まだ、ソフトウェアの実行作業とは連携していないため、実行時のビルド作業は行えないが、これをJavaの仮想マシンと連携させることにより、実行時のビルドプロセス管理を行うようにする予定である。

参考文献

[1] Bertino E. and Martino L. Object-Oriented Database Systems, Addison Wesley, 1993

[2] Cohen, E. S., D. A. Soni, R. Gluecker, H. W. M., R. W. Schwanke and M. E. Wagner. Version Management in Gypsy. in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development

Environment, pages 201-215, Boston, Massachusetts, ACM, November, 1988.

[3] Dittrich, K. R., W. Gotthard and P. C. Lockermann. DAMOKLES - A Database System for Software Engineering Environment,. in Proceedings of the International Workshop on Advanced Programming Environments, pages 353-371, Trondheim, Norway, IFIP WG 2.4, June, 1986.

[4] Estublier, J. and R. Casallas. The Adele Configuration Manager. In Configuration Management. Tichy ed. pages 99-133, John Wiley & Sons, 1994

[5] Feldman, S. I. MAKE - A Program for Maintaining Computer Programs. Software-Practice and Experience, vol. 9, pages 255-265, 1979.

[6] Kaiser, G. E. and A. N. Habermann. An Environment for System Version Control. in Proceedings of the COMPCON Spring 83, pages 415-420, IEEE, March, 1983.

[7] Kim, W. and Chou, H. T. Versions of Schema for Object-Oriented Databases, in Proceedings of the International Conference on Very Large Databases, Los Angeles, CA, August 1988

[8] Lampson, B. W. and E. E. Schmidt. Organizing Software in a Distributed Environment. in Proceedings of the ACM SIGPLAN 83 Symposium on Programming Language Issues in Software Systems, pages 1-13, ACM, 1983.

[9] Leblang, D. B., R. P. Chase Jr. and G. D. McLean Jr. The DOMAIN Software Engineering Environment for Large Scale Software Development Efforts. in Proceedings of the IEEE Conference on Workstations, pages 266-280, San Jose, California, IEEE, November, 1985.

[10] Rochkind, M. J. The Source Code Control System. IEEE Transactions on Software Engineering, vol.SE-1, no.4, pages 364-370, December, 1975.

[11] Sandberg, R., D. Goldberg, S. Kleinman, D. Walsh and B. Lyon. Design and Implementation of the Sun Network File System. in Proceedings of the Summer USENIX, pages 119-130, June, 1985.

[12] Sugiyama, Y. Producing and Managing Software Objects in the Process Programming Environment OPM. in Proceedings of APSEC94, pages 268-277, Tokyo, Japan, IEEE and IPSJ, 1994.

[13] Sugiyama, Y. Object Make: A Tool for Constructing Software Systems from Existing Software Components. in Proceedings of the Symposium on Software Reusability, pages 128-136, Seattle, ACM, April, 1995.

[14] 杉山安洋、プロセスオブジェクトの動的変更、オブジェクト指向'95シンポジウム論文集、pages 109-116、情報処理学会、1995年6月

[15] 杉山安洋、クライアントサーバ型コンフィギュレーション管理、ソフトウェアの新しい構成統合技術に関するワークショップ論文集、pages 39-48、日本ソフトウェア科学会、1996年3月

[16] Sun Microsystems, http://www.java.sun.com

[17] Sun Microsystems, SunOS 5.3 Linker and Libraries Manual

[18] Tichy, W. F. RCS - A system for Version Control. Software - Practice and Experience, vol.15, no.7, pages 637-654, July, 1985.

[19] Zdonik, S. B. Version Management in an Object-Oriented Database. in Proceedings of the International Workshop on Advanced Programming Environments, pages 405-422, Trondheim, Norway, IFIP WG 2.4, June, 1986.