プログラミング 8－9
（1996. 8. 28）

並行トランザクション機構の実装

大島芳樹　　脇田建

東京工業大学
数理・計算科学専攻
〒152 東京都目黒区大岡山 2-12-1
E-mail: {ohshima,wakita}@is.titech.ac.jp

並行プログラミング言語における並行性制御のプリミティブとして並行トランザクション機構は有用である。しかし、並行トランザクションの実装に関しては、プロトタイプの実装は試みられているものの、効率的な実装法は未だ確立されていない。我々は並行トランザクションの実装技法を新たにいくつか提案し、それを使用した並行トランザクションを実装し言語に導入した。この言語では、並行トランザクション機構のオーバーヘッドを小さく抑えることができた。

キーワード: 並行性制御、並行トランザクション、効率的実装

An Implementation of the Concurrent Transaction Mechanism

Yoshiki OHSHIMA　　Ken WAKITA

Tokyo Institute of Technology
Department of Mathematical and Computing Sciences
2-12-1 Ookayama, Meguro-ku, Tokyo, 152
E-mail: {ohshima,wakita}@is.titech.ac.jp

Abstract: The Concurrent transaction mechanism is an effective concurrency control mechanism for concurrent programming languages. Although there are several prototype implementations of concurrent transactions, efficient implementation is stil an ongoing research topic. This article proposes some novel implementation techniques and we implement the concurrent transactions mechanism using them and incorporate it into a concurrent language. In this language, we can keep the penalty of concurrent transactions at low level.

Keywords: Concurrency control, Concurrent transactions, efficient implementation

1 はじめに

並行プログラミング言語においては、並行性をいかにして生み出すかという問題と表裏一体に、複雑な並行性をいかにしてプログラマの制御下に置くかという並行性制御の問題がある。抽象度の高い並行性制御プリミティブであるトランザクション機構を言語に導入する試みはLiskovやDetlefsなどによって以前からなされているが[4, 2]、それらは並行性をかなり強く制限するものであった。この制限を緩和し、トランザクション内の並行性、入れ子トランザクション、非同期動作などを許した並行性制御機構として**並行トランザクション**がある。並行トランザクションはHARMONY/1 [7]、Venari/ML [3]などの並行言語に導入され、言語の記述力を大きく向上させることに成功している [6]。しかし、それらの言語はプロトタイプこそ実装されているものの、並行トランザクションの効率的な実装法は未だ確立されるには到っていない。

並行トランザクション機構をプログラミング言語に導入する際の問題は、データの操作を行うたびに発生するオーバーヘッドである。我々は、この問題に対処するため、効率の向上に有効ないくつかの実装技術を考案した。さらに、これらの技法を使用し、並行トランザクションを導入した並行オブジェクト指向言語HARMONY/2を実装した。

その結果、並行トランザクションを使用しないプログラムをHARMONY/2で動作させた場合、メソッド起動のオーバーヘッドをほとんどなくすことができ、メソッド実行のオーバーヘッドも11～40%程度に押さえることができた。トランザクション機構を利用するプログラムの場合は最大で4.7倍程度のオーバーヘッドとなった。

本稿の構成は以下の通りである。2節では、並行トランザクション機構の概要について述べる。3節では並行トランザクションの実装方針について述べる。4節では並行トランザクション機構を持つ並行オブジェクト指向言語HARMONY/2の実装について述べる。5節では実装されたHARMONY/2のプロトタイプに関する性能を評価する。6節では並行トランザクションの適用性などについて議論する。

2 並行トランザクションの概要

並行トランザクションは、分散データベースシステムなどで使用されているトランザクション機構を拡張した、並行言語の同期機構である。トランザクションとは逐次的に実行されるオブジェクトに対する操作(リードまたはライト)の列である [1]。この操作の列は、すべての操作が完全に実行され、その結果が永続的に保存されるか、あるいはなんらかの理由で失敗し、なんの影響も残さずに終わるかどちらかである。前者を**コミット**、後者を**アボート**という。この性質を保証するために、トランザクションは以下の4つの性質(***ASID*性**)を持つとされている。

原子性 (A) トランザクションは、すべての操作が完全に実行されるか、あるいはまったく実行されずに終わる(*all-or-nothing*)。

直列化可能性 (S) トランザクション同士はお互いの実行に影響を及ぼさない。すなわち複数のトランザクションが並行に実行されコミットした後の結果は、これらがある順序で逐次的に実行された結果と等しくなる。

孤立性 (I) あるトランザクション実行の途中経過は、他のトランザクションからは参照できない。

耐久性 (D) 一旦コミットをしたトランザクションの実行結果はシステムの故障があっても失われない。 □

一方、並行トランザクションにおいては、通常のリードおよびライト操作の他に、トランザクション内でのスレッド生成とトランザクション起動が許されている。トランザクション内でのトランザクション起動によってトランザクションの入れ子(**入れ子トランザクション** [5])が生じる。入れ子の内側のトランザクション(**子トランザクション**)の実行は外側のトランザクション(**親トランザクション**)の実行の一部である。そのため、並行トランザクションTの行った操作とは、T内のスレッドが行った操作および、Tの一部であるTの子孫すべての操作を合わせたものとなる。並行トランザクションがMossの入れ子トランザクションと異なっている点はトランザクションの親子同士が並行動作を行う点、およびトランザクション内部に並行性がある点である。

並行トランザクションの性質は、前述の$ASID$性を拡張した以下のような性質(***ASI*性**)を持つ。

原子性 (A) トランザクションが行った操作すべてに対して*all-or-nothing*の性質が保証される。

直列化可能性 (S) 親子関係にないトランザクション同士はお互いの実行に影響を及ぼしあわない。一方、親子関係にあるトランザクション同士の場合、子は親からトランザクションとして起動されたものなので、親からも影響を受けずに実行される。しかし子の動作は親の動作の一部なので、子の動作が親の動作に影響を及ぼすことが許されている。

孤立性 (I) Tの実行の途中経過は、親子関係にないトランザクションやTの祖先トランザクションからは参照できない。しかし、Tの子孫トランザクションからは参照することができる。 □

並行トランザクションはファイルシステム等からは独立した言語レベルの機能を意図しているので、耐久性に関しては考慮していない。

子孫を持つトランザクションがアボートするときは、子孫すべてを再帰的にアボートし、その後に自身がアボートする。一方、コミットしようとする場合は、自身の

部分である子孫のコミットを待ってからコミットする。並行トランザクションの操作的意味は [8] に与えられている。

3 実装の方針

以下の各節で、*ASI*性のそれぞれの性質をいかにして実現するかについて概略を述べる。

3.1 直列化可能性の実現法

直列化可能性は、Moss によって拡張された 2 相ロック方式 [5] によって実現することができる。[5] では従来の 2 相ロック方式と同様、オブジェクトごとにリードロックとライトロックを設けてトランザクションの排他的操作を表現する。あるトランザクションがオブジェクトに対してロックを掛けることは、そのオブジェクトに対して操作を行う権利を取得することを意味する。並行トランザクションの直列化可能性 (S) の後半、すなわち「子は親の一部なので子の動作が親の動作に影響を及ぼしても良い」という性質は、「親がロックを掛けていても、子がさらにロックを獲得して良い (**ロックの継承**)」という規則によって実現する。

並行性を過度に制限しないために、リードロックは共有ロックとする。ライトロックは基本的には排他ロックとするが、ロックの継承規則によって、ライトロックも多重に獲得されることがある。そのため、オブジェクト o のリードロックとライトロックはそれぞれトランザクションの集合となる。以下それぞれを $R(o)$ および $W(o)$ と表記する。

トランザクション T がオブジェクト o に対してリードまたはライト操作を行う場合には、まず以下の規則 C によってロックの整合性を検査する。そして、整合性が認められた場合にのみ $R(o)$ または $W(o)$ に T を追加し、実際の操作を行うことが許される。

C: C_R (リードの場合) $W(o)$ のすべての要素が T の祖先の場合のみリードが許される。

C_W (ライトの場合) $R(o)$ および $W(o)$ のすべての要素が T の祖先の場合にのみライトが許される。

規則 C に従うと、一般に $W(o)$ と $R(o)$ は図 1 のような構造になる。図の木構造は親子関係にあるトランザクションを表したものである。図中、黒く塗られたノードは $W(o)$ の要素、網掛けをされたノードは $R(o)$ の要素、点線で描かれたノードは o のロックの獲得が許されないトランザクション、白抜きのノードは o にアクセスしなかったトランザクションを表している。トランザクション T が $W(o)$ に入ることができるのは、$W(o)$ のすべての要素が祖先のときのみなので、$W(o)$ は全順序集合がなりたつ鎖となる (図中 a)。一方、いったん $W(o)$ の要素 l の下の複数の枝にある要素が $R(o)$ に入ってしまうと、$W(o)$ に新たな要素を追加することはできない。そのため、$R(o)$ には l の下にある部分木 (図中 b) の任意のトランザクションが入りうる。

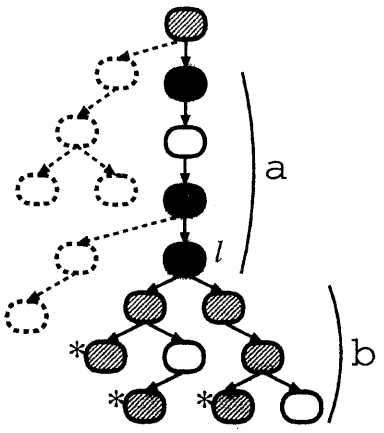

図 1 あるオブジェクトに掛けられているロックの状態

3.2 原子性の実現法

トランザクションの原子性を実現するために、あるトランザクション T がオブジェクト o を初めて変更するときには、o の変更前の状態を保存してから実際の変更を行う。バージョンの作成は $W(o)$ に新たなトランザクションが追加されるときに行われるので、$W(o)$ が鎖になるのと同様に、バージョンも複数の要素を持つ鎖状の構造となる。

T がアボートするときは、T がオブジェクトをまったく変更しなかったかのように振る舞わなくてはならない。そのために、T が変更したオブジェクトのそれぞれに対して、T が作成したバージョンをオブジェクトに書き戻すという処理を行う。

一方、T がコミットした場合、その後に T の親 P がアボートする可能性がある。その場合には P の原子性を保つために、T の行った変更も含めて復帰しなくてはならない。そのため、T がコミットした場合には T の行った変更を P の行った変更に変換する必要がある。ただし、P がすでに o を変更していた場合は (P の行った変更は必ず T の行った変更よりも古いので) バージョンを破棄して良い。

3.3 孤立性の実現法

多くの並行言語では、共有されているオブジェクトを通じての情報交換以外にも、スレッド同士がなんらかの情報交換を行う手段が提供されている。子トランザクシ

ョンから親トランザクションへの情報交換が行われると子の孤立性が失われるので、そのような情報交換に制限を加え、子トランザクションがコミットしたときのみ親トランザクションに情報が伝達されるようにする必要がある。

4 処理系の実装

並行トランザクションでは、オブジェクトに対する操作ごとにロックの検査を行う必要があるので、ロック検査の効率が並行トランザクション全体の性能を大きく左右する。また、バージョン作成のコストも大きな比重を占める。我々は、ロック操作の効率に大きな影響を与える**トランザクション識別子 (TID)** とロックに関して、新たな表現法を開発した。これらの表現法によって、トランザクションの親子関係の検査とロックの検査を定数時間で実行できるようになった。また、mutable なデータ構造に対して検討を加え、バージョン作成を浅いコピーのみで行えるようにした。

4.1 HARMONY/2

HARMONY/2 は Scheme をベースとし、それに Send & Reply 型のメッセージ送信を基本とするオブジェクト指向拡張を施したものである。HARMONY/2 はオブジェクト内に並行性のないモデルに基づいている。そのため、インスタンス変数ごとにロックを掛ける必要はなく、オブジェクト単位のロックで十分である。ロック獲得のタイミングは以下のようにする。ライトロックはオブジェクトの状態を変更するインスタンス変数への代入が起こったときとする。リードロックのタイミングは、「ライトロックに先だってリードロックを獲得しなくてはならない」という制限を加えるために、メッセージが受理されたときとする。

オブジェクト内に並行性がなく、リードロックはメッセージが受理されたときに獲得されるので、インスタンス変数を読み込む際にはロックを検査する必要がない。また、あるメソッド起動の中でいったんライトロックの獲得に成功した後では、2 度目以降の書き込みの際はロック検査を行う必要はない。

一般に、並行トランザクションを導入したアプリケーションでは、メッセージ送信やインスタンス変数への代入などの操作の回数と比較して、トランザクションの起動・終了の回数は非常に少ないことが普通である (トランザクションの希少性)。

4.2 トランザクション識別子の表現法

トランザクション識別子 (TID) とは、トランザクションごとに付加された一意性を表すための情報である。しかし、並行トランザクションでは、トランザクションの一意性だけではなく親子関係も検査できる必要がある。我々は、TID が木構造内での位置を示す情報も含むように拡張し、親子関係の検査は TID を使用して行う方法を考案した。

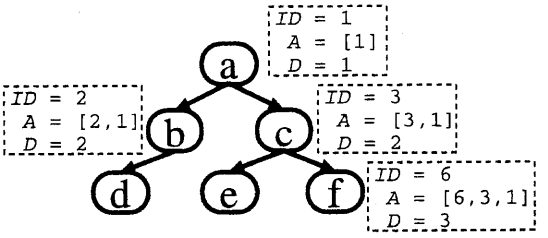

図 2 TID の表現

この方法では、木の各ノード Tに一意な整数$T.ID$と、そのノードの祖先のIDをルートに至るまで並べた配列$T.A$、およびトランザクションの深さを表す整数$T.D$を付加する。図 2はトランザクションの親子関係を木構造で表したものである。ノード f を例に取ると、f の IDは 6、f の A は、[f 自身、f の親 c、c の親 a] の IDの配列 $[6,3,1]$、f の Dは 3 となる。

これによって、TID の同値性の比較 ($=_t$と表記する) は、ID の比較、すなわち ($x.ID = y.ID$) で実現でき、親子関係の検査 $x >_t y$は、yから ($y.D - x.D$) 段上がった先祖が x であるかどうかの検査、すなわち、$x.ID = y.A[y.D - x.D]$ で実現できる。

図 2において $c >_t f$ を例に取ると、$f.A[3-2] = 3$ であり、これは $c.ID$と等しいので f は c の子孫であると言える。一方 $b >_t f$ を例とすると、$f.A[3-2] = 3 \neq 2$ なので、b と f の間には親子関係がないことが言える。

4.3 ロックの表現

3節の規則 Cは $R(o)$ と $W(o)$ がトランザクションの集合であるとして定義されていたが、実際にはこれらの集合は図 1のような特定の構造に制限されている。この制限を利用することによって、ロックの検査を高速化できる。以下で、まずこの制限から導ける最適化を列挙する。

1. $W(o)$ は鎖なので、C_Rは $W(o)$ 中の葉の要素 (図 1における lのみとの比較に簡略化できる。

2. ライトロックに先だってリードロックが獲得されているので、C_Wは $R(o)$ のみとの検査に簡略化できる。

3. C_Wは $R(o)$ の中の葉の要素 (図 1で "*" が付加された要素) との比較に簡略化できる。

4. 3. で葉の要素が複数存在する場合には、すでに $R(o)$ 内に Tの祖先でないトランザクションが存在することになるので、直ちに検査に失敗するすることが判る。

これらの性質から、規則 Cを以下のような規則 C'に改良できる。

C': C'_R (リードの場合) $W(o)$ の葉の要素が T の祖先の場合のみ許される。

C'_W (ライトの場合) ($R(o)$ の葉の要素数 $= 1$) かつ ($R(o)$ の葉の要素 $=_t T$) の場合に許される。

C' の実現においては、トランザクションの希少性を考慮して、$R(o)$ や $W(o)$ の更新のコストよりもロックの検査のコストを減少させることが重要である。C' では $R(o)$ や $W(o)$ の葉の要素のみを使用するので、$W(o)$ と $R(o)$ のどちらも葉の要素を常に追跡するようなデータ構造とする。

$W(o)$ の表現 $W(o)$ は鎖であり、葉の要素の追加・削除のみが行われる FILO 性があるデータ構造である。そこで、スタックを用いて実装する。

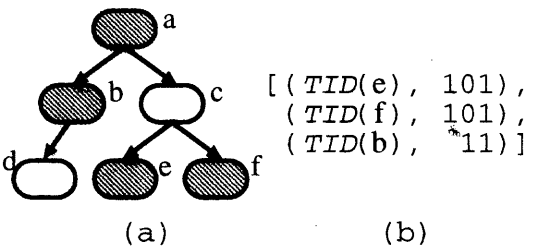

図 3 $R(o)$ の表現

$R(o)$ の表現 $R(o)$ は 2 つ組のリストとして表現する。組の第一要素は $R(o)$ の葉の要素のひとつであり、第二要素は、$R(o)$ 内にある第一要素の祖先の要素の存在を、ビット列に圧縮して表現したものである。図 3を例に取ると、図中 (a) の網掛けされたトランザクションが o にリードロックを掛けている場合、$R(o)$ は図中 (b) のような 2 つ組のリストとして表現される。ただし、TID(x) は ID が x であるような TID である。TID(f) と TID(e) の第二要素は、ともに自分自身と親の親が $R(o)$ に入っており、親が入っていないので 101 というビット列になっている。

この表現法の元では、C'_R は $(\mathrm{top}(W(o)) >_t T)$ として、C'_W は $(\mathrm{length}(R(o)) = 1 \wedge \mathrm{car}(\mathrm{car}(R(o))) =_t T)$ として、どちらも数回の演算を行うだけで実行できる。ただし、top() はスタックの先頭の要素を返す関数であり、car() はリストまたは 2 つ組の先頭の要素を返す関数である。

4.4 mutable なデータ構造への対処

オブジェクトのバージョンを作成する場合には、データの浅いコピーのみで行なえることが望ましい。しかし、mutable なデータ構造が存在する場合、浅いコピーを行うだけでは、あるオブジェクトのバージョンとして作成しておいたデータが予期せぬ変更を受けてしまい、トランザクションの原子性が正しく保たれなくなる可能性がある。

この問題は、(a) mutable なデータ構造を禁止する、(b) mutable なデータ構造は HARMONY/2 のオブジェクトとして実装し直す、(c) 深いコピーを行う、のいずれかの方法によって解決できる。

HARMONY/2 がベースとしている Scheme では、cons セル、string、vector の 3 つが mutable なデータ構造である。cons セル (リスト) は Scheme の基本的なデータ構造であり、これを HARMONY/2 のオブジェクトとして実装し直すのはペナルティが大きすぎる。そこで、HARMONY/2 では、(a)、(b) の方法を組合せ、cons セルに対する破壊的操作 (`set-car!` と `set-cdr!`) を禁止して cons セルを immutable なものとし、string と vector を HARMONY/2 のオブジェクトとして実装している。この制限によって、オブジェクトのバージョンを浅いコピーで行えるようになる。

表 1 HARMONY/0 と HARMONY/2 の比較

例題名	H0 (秒)	H2 (秒)	$H0/H2$
minimum (100000)	2.57	2.53	1.02
empty (100000)	3.19	3.55	1.11
create (10000)	7.64	9.46	1.24
write (10000)	0.35	0.49	1.40

表 2 トランザクションを使用した例

例題名	実行時間	表 1との比
empty (100000)	16.65	4.69
write (10000)	1.10	2.24

5 評価

以下で HARMONY/2 の性能評価の結果を挙げる。HARMONY/2 のプロトタイプは Scheme を用いて実装されている。処理系には Gambit-c 2.3 を使用した。計測に使用したマシンは SPARC Station 10 互換機である。

まず、トランザクションを使用しない例を用いて、HARMONY/2 と、HARMONY/2 から並行トランザクションの機能を除いた処理系 HARMONY/0 とを比較する。これによって並行トランザクション機構のオーバーヘッドが計測できる。この結果を表 1に挙げる。表中、H0 の欄は HARMONY/0 の数値を、H2 は HARMONY/2 の数値を表している。

HARMONY/2 では、インスタンス変数を持たないオブジェクトのメソッドではロックの検査を一切行わないようにしている。minimum はこのようなオブジェクトに対して中身のないメソッドを 100000 回呼び出すとい

うものである。この場合、基本的にはHARMONY/0とHARMONY/2で同じコードが実行されるので、オーバーヘッドはほとんどない。

emptyは100000回のメッセージ送信を空のメソッドに対して行うものである。HARMONY/2のほうはHARMONY/0には存在しないロック検査ルーチンを呼び出すので、これが11%のオーバーヘッドになっている。ただし、トランザクションが存在しない場合には、HARMONY/2はロック検査ルーチンを空の関数で置き換えるので、オーバーヘッドが小さく抑えられている。

createはオブジェクトの生成を10000回行うというものである。24%のオーバーヘッドはロックを表現するためのデータ構造を生成することによる。

writeはインスタンス変数に書き込みを行うメソッドを10000回呼び出すものである。HARMONY/0ではインスタンス変数への書き込みは必ず成功するが、HARMONY/2では書き込み時にロックが獲得できずブロックする可能性がある。40%のオーバーヘッドはブロックに対処するためのコードによるものである。

次に、トランザクションを使用した例を表2に挙げる。この例は、あるトランザクションによってライトロックが獲得されているオブジェクトに対して、emptyおよびwriteと同様の処理を行ったものである。表2のemptyではロック検査、$R(o)$への要素の追加、トランザクションのリードセットへのオブジェクトの追加が4.69倍のオーバーヘッドの原因である。$R(o)$はやや複雑な構造をしているため、これへの要素の追加もある程度のコストがかかる。トランザクションを使用していない場合のempty(H2)ではあまり多くの処理を行っていないため、これらの処理を行うだけでも倍率としては大きな数値となってしまう。ただし、4.69という数値はトランザクション機構にもっとも不利に働くベンチマークによるものであり、起動されるメソッドがある程度の大きさであれば、そのメソッドの大きさに反比例して影響は小さくなる。また、同じトランザクションが同じオブジェクトに何度かアクセスする場合もオーバーヘッドは小さくなる。

writeに関してはロックの検査、バージョンの作成、ライトセットへの追加がオーバーヘッドとなっている。こちらも一度だけインスタンス変数に代入して返るというごく小さいメソッドを使用したために倍率としては大きなものになっている。

6 おわりに

5節で示したように、トランザクションが起動されていない場合は、メッセージ送信時のオーバーヘッドは11%程度、インスタンス変数への書き込みは40%程度とそれほど大きくはない。ただし、4.1節で述べたように、リードロックとライトロックの検査はメソッド起動ごとに一回行えばよい。現実的なアプリケーションでは、ほとんどのメソッドは複数回インスタンス変数を操作する。そのため、並行トランザクションを持たない言語と比べてもオーバーヘッドはほとんどないと言える。

トランザクション機構にもっとも負荷を掛けた場合のオーバーヘッドは、4.69倍という値になってしまった。今後は、このオーバーヘッドを低減する必要がある。その方針としては以下のようなものが考えられる。

C言語の利用 トランザクション機構のランタイムモジュールをC言語で記述する。C言語の持つビット操作の機能や高速な配列などを使用して、$R(o)$の表現で使用しているビット操作やTIDの操作を高速化する。

メソッドの分類 インスタンス変数への操作パターンによってメソッドを分類し、インスタンス変数を操作しないメソッドに対してはロックの検査を省略する。

トランザクションローカルなオブジェクト あるトランザクションの内部で生成され、トランザクション外部にはその存在が伝えられないオブジェクトに関してはロックの検査やバージョンの作成を省略する。

参考文献

[1] S. Ceri and G. Pelagatti. *Distributed Databases: Principles & Systems*. McGrawHill, New York, 1985.

[2] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *IEEE Computer*, 21(12):57–69, 1988.

[3] N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing first-class transactions. *Transactions on Programming Languages and Systems*, 16(6):1719–1736, 1994.

[4] B. Liskov. Distributed programming in Argus. *Commun. ACM*, 31(3):300–312, 1988.

[5] J. E. B. Moss. *Nested Transaction: An Approach to Reliable Distributed Computing*. The MIT Press, Cambridge, Massachusetts, 1985.

[6] K. Wakita and Y. Ohshima. Concurrent transactions and communicators: Extensible synchronization mechanisms for distributed programming. Technical Report C-118, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 1995.

[7] K. Wakita and A. Yonezawa. Linguistic supports for development of distributed organizational information systems in object-oriented concurrent computation frameworks. In *ACM Conference on Organizational Computing Systems*, pages 185–198, November 1990.

[8] Ken Wakita. Semantics of concurrent transactions. In *Proceedings of JSSST Workshop on Object-Oriented Computing*, 1996. URL:http://www.rwcp.or.jp/ people/ishikawa/wooc96/wooc96.html.