(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号
特許第4739465号
(P4739465)
(45)発行日　平成23年8月3日(2011.8.3)　(24)登録日　平成23年5月13日(2011.5.13)

(51)Int.Cl.　F I
G06F 21/22 (2006.01)　G06F 9/06 660L

請求項の数 6　(全 50 頁)

(21)出願番号	特願平11-508660
(86)(22)出願日	平成10年6月9日(1998.6.9)
(65)公表番号	特表2002-514333(P2002-514333A)
(43)公表日	平成14年5月14日(2002.5.14)
(86)国際出願番号	PCT/US1998/012017
(87)国際公開番号	WO1999/001815
(87)国際公開日	平成11年1月14日(1999.1.14)
審査請求日	平成17年6月9日(2005.6.9)
(31)優先権主張番号	328057
(32)優先日	平成9年6月9日(1997.6.9)
(33)優先権主張国	ニュージーランド(NZ)

(73)特許権者
インタートラスト　テクノロジーズ　コーポレイション
アメリカ合衆国，カリフォルニア　９４０８６，サニーベール，オークミード　パークウエイ　４６０
(74)代理人
弁理士　石田　敬
(74)代理人
弁理士　鶴田　準一
(74)代理人
弁理士　下道　晶久
(74)代理人
弁理士　西山　雅也

最終頁に続く

(54)【発明の名称】ソフトウェアセキュリティを増強するための混乱化技術

(57)【特許請求の範囲】
【請求項１】
コンピュータ・コードを混乱化するためにコンピュータによって実施される方法であって、
混乱化器が、混乱化すべきコンピュータ・コードを記憶ユニットにロードするステップと、
該混乱化器が、ロードされた該コンピュータ・コードの内部データ構造をグラフによって構築し、該グラフによって内部データ構造が構築された該コンピュータ・コードに対し、所与の混乱化レベルに達し終わるまで、１つ以上の混乱化変換を選択して適用することにより、混乱化されたコンピュータ・コードを生成するステップであって、少なくとも１つの混乱化変換が、
１つのアレー（array）を少なくとも２つのアレーに分割することを含む変換；
２つのアレーを単一のアレーに併合することを含む変換；
１つのアレーを異なる次元に最構成することを含む変換；
少なくとも１つのループを分割することを含む変換；
２つの変数を１つの変数に併合することを含む変換；
１つの変数を少なくとも２つの変数に分割することを含む変換；及び
少なくとも１つのストリングを、該ストリングを生成する手続き（procedure）呼び出しで置き換えることを含む変換；
のうちの一つであるステップと、

を含む方法。
【請求項２】
コンピュータ・プログラム又はモジュールを混乱化するためにコンピュータによって実施される方法であって、該コンピュータ・プログラム又はモジュールは第一のローカル変数を含む第一のプロシージャー及び第二のローカル変数を含む第二のプロシージャーを含み、該方法が、
混乱化器が、該コンピュータ・プログラム又はモジュールを記憶ユニットにロードするステップと、
該混乱化器が、ロードされた該コンピュータ・プログラム又はモジュールの内部データ構造をグラフによって構築し、該グラフによって内部データ構造が構築された該コンピュータ・プログラム又はモジュールの少なくとも一部分に対し、所与の混乱化レベルに達し終わるまで、変更を選択して実行することにより、混乱化されたコンピュータ・プログラム又はモジュールを生成するステップであって、該変更が、
グローバル変数を生成すること；
該第一の変数への少なくとも１つの参照を該グローバル変数への参照によって置き換えること；及び
該第二の変数への少なくとも１つの参照を該グローバル変数への参照によって置き換えること；
を含むステップと、
を含む方法。
【請求項３】
コンピュータ・プログラム又はモジュールを混乱化するためにコンピュータによって実施される方法であって、該コンピュータ・プログラム又はモジュールは１つ以上の定められたタスクを実行するように設計され第一のスレッドを含み、該方法が、
混乱化器が、該コンピュータ・プログラム又はモジュールを記憶ユニットにロードするステップと、
該混乱化器が、ロードされた該コンピュータ・プログラム又はモジュールの内部データ構造をグラフによって構築し、該グラフによって内部データ構造が構築された該コンピュータ・プログラム又はモジュールの少なくとも一部分に対し、所与の混乱化レベルに達し終わるまで、変更を選択して実行することにより、混乱化されたコンピュータ・プログラム又はモジュールを生成するステップであって、該変更が、
第二のスレッドを生成すること；及び
該コンピュータ・プログラム又はモジュールに１つ以上のプログラミング・ステートメントを挿入することであって、該プログラミング・ステートメントは該１つ以上の定められたタスクに寄与する機能を何も実行せず、該プログラミング・ステートメントの１つ以上は該第二のスレッドで実行するように設計されること、
を含むステップと、
を含む方法。
【請求項４】
コンピュータ・プログラム又はモジュールを混乱化するためにコンピュータによって実施される方法であって、該コンピュータ・プログラム又はモジュールは第一の文字列を含み、該方法が、
混乱化器が、該コンピュータ・プログラム又はモジュールを記憶ユニットにロードするステップと、
該混乱化器が、ロードされた該コンピュータ・プログラム又はモジュールの内部データ構造をグラフによって構築し、該グラフによって内部データ構造が構築された該コンピュータ・プログラム又はモジュールの少なくとも一部分に対し、所与の混乱化レベルに達し終わるまで、変更を選択して実行することにより、混乱化されたコンピュータ・プログラム又はモジュールを生成するステップであって、該変更が、
該第一の文字列をダイナミックに生成するように設計されたプログラミング・コンストラ

クトを生成すること；及び
該第一の文字列の少なくとも１つのインスタンスを該プログラミング・コンストラクト又は該プログラミング・コンストラクトへの呼び出しによって置き換えること；
を含むステップと、
を含む方法。
【請求項５】
コンピュータ・プログラム又はモジュールを混乱化するためにコンピュータによって実施される方法であって、該コンピュータ・プログラム又はモジュールは第一のプロシージャーと第二のプロシージャーとを含み、該方法が、
混乱化器が、該コンピュータ・プログラム又はモジュールを記憶ユニットにロードするステップと、
該混乱化器が、ロードされた該コンピュータ・プログラム又はモジュールの内部データ構造をグラフによって構築し、該グラフによって内部データ構造が構築された該コンピュータ・プログラム又はモジュールの少なくとも一部分に対し、所与の混乱化レベルに達し終わるまで、変更を選択して実行することにより、混乱化されたコンピュータ・プログラム又はモジュールを生成するステップであって、該変更が、
第三のプロシージャーを生成することであって、該第三のプロシージャーが、
該第一のプロシージャーの少なくとも一部分；
該第二のプロシージャーの少なくとも一部分；
該第一のプロシージャーのパラメータ・リスト；
該第二のプロシージャーのパラメータ・リスト；及び
該第三のプロシージャーへの呼出しが該第一のプロシージャー又は該第二のプロシージャーの実行を指定できるように設計されたプログラミング・コンストラクト；
を含むものと、
少なくとも１つの該第一のプロシージャーへの呼出しを該第三のプロシージャーへの呼出しによって置き換えることであって、該第三のプロシージャーへの呼出しは該プログラミング・コンストラクトが該第三のプロシージャーに該第一のプロシージャーの少なくとも一部分を実行させるために用いる情報を含むものと、
を含むステップと、
を含む方法。
【請求項６】
コンピュータ・プログラム又はモジュールを混乱化するためにコンピュータによって実施される方法であって、該コンピュータ・プログラム又はモジュールは少なくとも第一のクラスを含み、該方法が、
混乱化器が、該コンピュータ・プログラム又はモジュールを記憶ユニットにロードするステップと、
該混乱化器が、ロードされた該コンピュータ・プログラム又はモジュールの内部データ構造をグラフによって構築し、該グラフによって内部データ構造が構築された該コンピュータ・プログラム又はモジュールの少なくとも一部分に対し、所与の混乱化レベルに達し終わるまで、変更を選択して実行することにより、混乱化されたコンピュータ・プログラム又はモジュールを生成するステップであって、該変更が、
第二のクラスと第三のクラスを生成することであって、該第三のクラスは該第二のクラスから直接継承し（inheriting）、該第二のクラスと該第三のクラスは該第一のクラスを置き換えるように設計され、
該第二のクラスと該第三のクラスを該コンピュータ・プログラム又はモジュールに組み込むこと、及び
該第一のクラスを該コンピュータ・プログラム又はモジュールから除去すること、
を含むステップと、
を含む方法。
【発明の詳細な説明】

発明の分野
本発明は、ソフトウェアの解釈、復号又はリバースエンジニアリングを防止又は少なくとも妨害するための方法及び装置に関する。より特定的に言うと、排他的ではないものの、本発明は、ソフトウェアから識別可能な構造又は情報を、逆コンパイル又はリバースエンジニアリングプロセスがさらに困難になるような形で挿入し、除去し又は再配置することによって、ソフトウェアの構造的かつ論理的複雑性を増大させるための方法及び装置に関する。
発明の背景
ソフトウェアは、その性質上第３者により分析されコピーされ易い。これまでにもソフトウェアセキュリティを増強させるために多大な努力が払われてきており、これらの成功はさまざまなものであった。このようなセキュリティの問題は、ソフトウェアの無許可コピーを防止する必要性及びリバースエンジニアリングを介して決定できるよりなプログラミング技術を隠したいという願望に関するものである。
例えば著作権の様な確立された法的方法は、立法上の保護措置を提供する。しかしながら、このような制度下で作り出された法的権利を主張することは、費用及び時間が共にかかる作業でありうる。さらに、著作権の下でソフトウェアに対し付与される保護は、プログラミング技術をカバーしていない。かかる技術（すなわち、ソフトウェアの形態に反する機能）は、法律上保護するのが困難である。リバースエンジニアは、問題のソフトウェアの機能の詳細な知識に基づき、最初から関連するソフトウェアを書直すことによって権利侵害から免れることができる。かかる知識は、データ構造、抽象化及びコードの組織を分析することから導き出すことができる。
ソフトウェア特許は、より広範な保護を提供してくれる。しかしながら、ソフトウェアの法的保護を技術的保護と結合させることは明らかに有利である。
所有権主張可能なソフトウェアの保護に対する従来のアプローチは、暗号化に基づいたハードウェア上の解決法を使用するか又は、ソースコード構造の単純な再配置に基づくもののいずれかであった。ハードウェアに基づく技術は、それが一般に費用のかかるものであり、特定のプラットフォーム又はハードウェアアドオンに結びつけられるものであるという点で、理想的ではない。ソフトウェアによる解決法は、標準的に、Java™用のCrema混乱化器といったトリビアルコード混乱化器を内含する。一部の混乱化器は、アプリケーションの語い構造を目標とし、通常ソースコードフォーマッティング及びコメントを除去し、変数を再命名する。しかしながら、このような混乱化技術は、悪意あるリバースエンジニアリングに対する充分な保護を提供しない。すなわち、リバースエンジニアリングは、ソフトウェアが分散される形とは無関係の問題である。さらに、ソフトウェアが、オリジナルソースコード内の情報の多く又は全てを保持するハードウェア依存性フォーマットで分散されている場合、問題はさらに悪化する。かかるフォーマットの例としては、Java™バイトコード及びアーキテクチュアニュートラル分散フォーマット（ＡＮＤＦ）がある。
ソフトウェア開発には、プログラマが多大な時間、努力そして技能を投入している可能性がある。商業的には、所有権主張可能な技術を競合者がコピーするのを防止できることは、きわめて重要である。
発明の開示
本発明は、リバースエンジニアリングに対するソフトウェアの抵抗力を高めるため（又は一般大衆に対し有用な選択肢を提供するため）のコンピュータにより実施される方法といったような、ソフトウェアセキュリティ用の混乱化技術のための方法及び装置を提供する。一実施形態においては、コードを混乱化するためにコンピュータにより実施される方法には、１個またはそれ以上のコードに対する混乱化変換の供給の完了についてテストする段階、混乱化すべきコードのサブセットを選択する段階、適用すべき混乱化変換を選択する段階、変換を適用する段階そして完了テスト段階へと復帰する段階が内含される。
１つの変形実施形態においては、本発明は、コンピュータ上で実行され、記憶され、又はそれにより操作されるソフトウェアが、予め定められ制御された程度のリバースエンジニアリング耐性を示すような形でコンピュータを制御する方法において、ソフトウェアの選

択された部分に選択された混乱化変換を適用する段階であって、必要とされる程度のリバースエンジニアリング耐性、ソフトウェアのオペレーションにおける有効性及び変換されたソフトウェアのサイズを提供するように選択された混乱化を用いて一定のレベルの混乱化を達成する段階；及び混乱化変換を反映するべくソフトウェアを更新する段階を内含する方法に関する。

好ましい実施形態においては、本発明は、ソフトウェアのセキュリティを増強するためのコンピュータにより実施される方法において、処理すべきアプリケーションのためソースソフトウェアに対応する１個またはそれ以上のソースコード入力ファイルを識別する段階；必要とされる混乱化レベル（例えば効力）を選択する段階；最大実行時間又は空間ペナルティ（例えばコスト）を選択する段階；任意にはソースコードにより直接又は間接的に読取られた任意のライブラリ又は補足的ファイルと共に入力ファイルを読取りかつ構文解析する段階；処理すべきアプリケーションによって使用されるデータタイプ、データ構造、及び制御構造を識別する情報を提供し、この情報を記憶するべく適切なテーブルを構築する段階、前処理段階に応えてアプリケーションについての情報を前処理する段階、ソースコードオブジェクトに対して混乱化コード変換を選択し適用する段階；必要とされる効力が達成されるか又は最大コストを上回ってしまうまで混乱化コード変換段階を反復する段階、及び変換済みソフトウェアを出力する段階を含んで成る方法を提供する。

好ましくは、アプリケーションに関する情報は、さまざまな静的分析技術及び動的分析技術を用いて得られる。静的分析技術としては、手順間データフロー分析及びデータ従属性分析が含まれる。動的分析技術としては、プロファイリングが含まれ、任意には、ユーザを介して情報を得ることができる。プロファイリングは、特定のソースコードオブジェクトに対し適用することが可能な混乱化レベルを決定するのに使用することができる。変換は、複数の不明瞭コンストラクト（構文）を使用して作成された制御変換を含むことができる。この不明瞭コンストラクトは、性能の見地から見て実行するのに廉価で混乱化器が簡単に構築できしかも混乱解除器がそれを破断するのには高価である、何れかの数学的オブジェクトである。好ましくは、不明瞭コンストラクトは、別名化及び同時実行技術を用いて構築されうる。ソースアプリケーションに関する情報は同様に、そのアプリケーションが含有するプログラム用イディオム及び言語コンストラクトの性質を決定するプラグマティック（実用）分析を用いて得ることができる。

混乱化変換の効力は、ソフトウェア複雑性メトリックを用いて評価可能である。混乱化コード変換は、あらゆる言語コンストラクトに応用できる。例えば、モジュール、クラス又はサブルーチンを分割又は併合することができ；新しい制御及びデータ構造を作成することもでき；又、オリジナル制御及びデータ構造を修正することもできる。好ましくは、変換されたアプリケーションに対し付加される新しいコンストラクトは、前処理の間に収集したプラグマティック情報に基づいて、ソースアプリケーション内のものにできるかぎり類似したものとなるように選択される。このメソッドは、混乱化変換がそれについて適用された情報及びソースソフトウェアに対する変換済みアプリケーションの混乱化されたコードに関する情報を含む補助ファイルを生成することができる。

好ましくは、混乱化変換は、Ｐが未変換ソフトウェアであり、Ｐ’が変換済みソフトウェアである場合、ＰとＰ’が同じ可観測性挙動を有するような形で、ソフトウェアの可観測性挙動を保つように選択される。より特定的に言うと、Ｐが終結できないか又はエラー条件を伴って終結した場合、Ｐ’は終結してもしなくてもよく、そうでなければＰ’は終結してＰと同じ出力を生成する。可観測性挙動としては、ユーザが経験する効果が含まれるが、Ｐ及びＰ’は、ユーザにとって観測不能な異なる詳細な挙動を伴って走行することができる。例えば、異なるものでありうるＰ及びＰ’の詳細な挙動としては、ファイル作成、メモリ使用及びネットワーク通信が含まれる。

一実施形態においては、本発明は同様に、スライシング、部分評価、データフロー分析又は統計分析を用いることにより、混乱化済みアプリケーションから混乱化を除去するために採用される混乱解除用ツールを提供する。

【図面の簡単な説明】

以下に、単なる一例として、図面を参照して本発明を説明する。
図１は、本発明の教示に従ったデータ処理システムを示す。
図２は、混乱化変換のカテゴリを含むソフトウェア保護の分類を例示する。
図３ａ及び３ｂは、（ａ）サーバー側実行及び（ｂ）部分的サーバ側実行によりソフトウエアセキュリティを提供するための技術を示す。
図４ａ及び４ｂは、（ａ）暗号化及び（ｂ）署名されたネーティブコードを用いることによりソフトウエアセキュリティを提供するための技術を示す。
図５は、混乱化を通してソフトウエアセキュリティを提供するための技術を示す。
図６は、Java™アプリケーションと共に使用するのに適した混乱化ツールの一例のアーキテクチャを例示する。
図７は、既知のソフトウエア複雑性メトリックセレクションを作表したテーブルである。
図８ａ及び８ｂは、混乱化変換の弾力性を例示している。
図９は、異なるタイプの不明瞭述語を示す。
図１０ａ及び１０ｂは、（ａ）トリビアルな不明瞭コンストラクト及び（ｂ）弱い不明瞭コンストラクトの例を提供している。
図１１は、計算変換（分岐挿入変換）の一例を示す。
図１２ａ～１２ｄは、ループ条件挿入変換を例示している。
図１３は、可約フローグラフを非可約フローグラフに変換する変換を例示する。
図１４は、データ従属性を全く含まない場合に、コード区分を並列化できることを示している。
図１５は、データ従属性を全く含まないコード区分を、適切な同期化基本命令を挿入することによって並行スレッドへと分割できることを示している。
図１６は、いかにして手順Ｐ及びＱがその呼出しサイトでインラインにされ、次にコードから除去されるかを示す。
図１７は、インライン処理メソッド呼出しを例示している。
図１８は、同じクラス内で宣言された２つのメソッドをインタリーブするための技術を示す。
図１９は、オリジナルコードに対して異なる混乱化変換セットを適用することにより１つのメソッドの複数の異なるバージョンを作成するための技術を示す。
図２０ａ～２０ｃは、（ａ）ループブロック化、（ｂ）ループアンローリング及び（ｃ）ループ分裂を含むロール変換の例を提供している。
図２１は、可変的分割例を示す。
図２２は、ストリング「ＡＡＡ」，「ＢＡＡＡＡ」及び「ＣＣＢ」を混乱化するように構築された１つの関数を提供している。
図２３は、２つの３２ビット変数ｘ及びｙを１つの６４ビット変数ｚへと併合する例を示す。
図２４は、アレイ再構成のためのデータ変換の例を示す。
図２５は、継承階層の修正を例示する。
図２６は、オブジェクト及び別名から構築された不明瞭述語を例示する。
図２７は、スレッドを用いた不明瞭コンストラクトの一例を提供している。
図２８ａ～図２８ｄは、（ａ）が、混乱化中の３つの文S_{1-3}を含むオリジナルプログラムを示し、（ｂ）が「恒常な」不明瞭述語を識別する混乱解除器を示し、（ｃ）が文の中の共通コードを決定する混乱解除器を示し、（ｄ）が、いくつかの最終的単純化を適用しプログラムをそのオリジナル形態に戻す混乱解除器を示している、混乱化対混乱解除の関係を例示する図である。
図２９は、Java™混乱解除ツールのアーキテクチャを示す。
図３０は、評価に使用される統計分析例を示す。
図３１ａ及び３１ｂは、さまざまな混乱化変換の概要のテーブルを提供する。
図３２は、さまざまな不明瞭コンストラクトの概要を提供する。

発明の詳細な説明

以下の記述は、出願人が現在開発中であるJava™混乱化ツールに関連して提供されるものである。ただし、当業者であれば、当該技術がその他のプログラミング言語にも適用可能であることは明白であり、本発明はJava™アプリケーションに制限されるものとみなされるべきではない。本発明のその他のプログラミング言語に関連しての実施は、当業者の視野内に入るものとみなされる。以下の実施例は、明確さを期して、特定的に、Java™混乱化ツールを目標としている。

以下の記述においては、次のような名称が用いられることになる。すなわち、Ｐは混乱化されるべき入力アプリケーションである；Ｐ’は変換済みアプリケーションである；Ｔは、それがＰをＰ’へと変換するような形での変換である。Ｐ（Ｔ）Ｐ’は、Ｐ及びＰ’が同じ可観測性挙動を有する場合の混乱化変換である。可観測性挙動は一般に、ユーザが経験する挙動として定義される。かくして、Ｐ’は、ユーザがそれを体験しないかぎりにおいて、Ｐがもたないファイルを作成するといったような予期せぬ効果を有する可能性がある。ＰとＰ’は必ずしも同等の効率をもつ必要はない。

ハードウェア例

図１は、本発明の教示に従ったデータ処理システムを例示する。図１は、３つの主要な要素を含むコンピュータ１００を示す。コンピュータ１００には、このコンピュータのその他の部分へ及びこの部分から適切に構造化された形態で情報を通信するのに使用される入出力（Ｉ／Ｏ）回路１２０が内含されている。コンピュータ１００には、Ｉ／Ｏ回路１２０及びメモリ１４０（例えば揮発性及び不揮発性メモリ）と通信状態にある制御処理ユニット（ＣＰＵ）１３０が内含されている。これらの要素は、大部分の汎用コンピュータに標準的に見られるものであり、実際、コンピュータ１００は、広範なカテゴリのデータ処理デバイスを代表するものとなるよう意図されている。ラスター表示モニター１６０がＩ／Ｏ回路１２０と通信状態で示され、ＣＰＵ１３０が生成する画素を表示するように命じられている。任意の周知のさまざまな陰極線管（ＣＲＴ）又はその他のタイプの表示装置を、表示装置１６０として使用することができる。従来のキーボード１５０も、Ｉ／Ｏ１２０と通信状態で示されている。当業者であれば、コンピュータ１００が、より大きいシステムの一部でありうることが理解される。例えば、コンピュータ１００は、１つのネットワーク（例えばローカルエリアネットワーク（ＬＡＮ）に接続されたもの）と通信状態にあってもよい。

特に、コンピュータ１００は、本発明の教示に従ってソフトウエアセキュリティを増強するための混乱化回路を内含することができ、或いは又、当業者であればわかるように、本発明をコンピュータ１００により実行されるソフトウェアの形で実施することも可能である（例えばこのソフトウェアをメモリ１４０内に格納してＣＰＵ１３０上で実行することができる）。例えば、メモリ１４０内に格納された未混乱化プログラムＰ（例えばアプリケーション）を、本発明の１実施形態に従ってメモリ１４０内に記憶された混乱化済みプログラムＰ’を提供するべくＣＰＵ１３０上で実行する混乱化器により、混乱化させることが可能である。

詳細な記述の概要

図６は、Java™混乱化器のアーキテクチャを示す。本発明の方法に従うと、Java™アプリケーションクラスファイルは、任意のライブラリファイルと共にパスされる。継承ツリーがシンボルテーブルと共に構築され、全てのシンボルについてのタイプ情報及び全てのメソッドについての制御フローグラフを提供する。ユーザは、任意には、Java™プロファイリングツールによって生成されるように、プロファイリングデータファイルを提供することができる。この情報は、アプリケーションのうちの頻繁に実行される部分が非常に高価な変換によって混乱化されていないことを保証するべく混乱化器を案内するのに使用できる。手順間データフロー分析及びデータ従属性分析といったような標準コンパイラ技術を用いてアプリケーションについての情報が収集される。その中には、ユーザにより提供されるものもあれば、専門的技術によって提供されるものもある。この情報は、適切なコード変換を選択し適用するために使用される。

適切な変換が選択される。大部分の適切な変換を選択する上で使用される支配的な基準は

、選択された変換がコードの残りの部分と自然に混ざり合うという必要条件を内含している。これは、高い適切性値をもつ変換を奨励することによって対処できる。もう１つの必要条件は、低い実行時間ペナルティで高レベルの混乱化を生み出す変換を奨励すべきであるというものである。後者の点は、効力及び弾力性を最大にしコストを最小限にする変換を選択することによって達成される。

混乱化の優先性が、ソースコードオブジェクトに割当てられる。これは、ソースコードオブジェクトの内容を混乱化することがいかに重要であるかを反映することになる。例えば、特定のソースコードオブジェクトが非常に感応性の高い所有権主張できる材料を含む場合、混乱化優先性が高くなる。各メソッドについて、実行時間ランクが決定され、これは、そのメソッドを実行するのに他のどれよりも多くの時間が費された場合１に等しい。

このとき、アプリケーションは、適切な内部データ構造、適切な変換への各ソースコードオブジェクトからのマッピング、混乱化優先性及び実行時間ランクを打ち立てることによって、混乱化される。混乱化変換は、必要とされる混乱化が達成されるか又は最大実行時間ペナルティを上回ってしまうまで適用される。変換済みアプリケーションは、この時点で書き込まれる。

混乱化ツールの出力は、機能的にオリジナルと等価である新しいアプリケーションである。このツールは、変換がそれについて適用された情報及び混乱化されたコードがいかにオリジナルアプリケーションと関連するかの情報が注釈として付いたJava™ソースファイルをも生成することができる。

ここで、ひきつづきJava™混乱化器に関連して、いくつかの混乱化変換例について記述する。

混乱化変換は、その質に従って評価し分類することができる。変換の質は、その効力、弾力性及びコストに従って表現できる。変換の効力は、Ｐ’がＰとの関係においていかにあいまいであるかに関係する。このようなメトリックは全て、必然的に人間の認識能力によって左右されることから、比較的不確かなものとなる。当該目的のためには、その変換の有用性の一尺度として変換の効力を考慮するだけで充分である。変換の弾力性は、変換が自動混乱解除器からの攻撃に対しいかにうまく持ちこたえるかを測定する。これは、プログラマの努力と混乱解除器の努力という２つの因子の組合せである。弾力性は、トリビアルからワンウェイまでの目盛上で測定できる。ワンウェイ変換は、それを逆転させることができないという点で極端なものである。第３の構成要素は、変換実行コストである。これは、変換済みアプリケーションＰ’を使用した結果としてこうむった実行時間又は空間ペナルティである。変換評価のさらなる詳細については、以下の好ましい実施形態の詳細な説明の部分で論述される。混乱化変換の主要な分類は、図２Ｃに示され、詳細は図２ｅ～２ｇに与えられている。

混乱化変換の例は、以下の通りである：混乱化変換は、制御混乱化、データ混乱化、レイアウト混乱化及び予防混乱化にカテゴリー分類されうる。これらのいくつかの例について以下で論述する。

制御混乱化には、集合変換、オーダリング変換及び計算変換が含まれる。

計算変換には、不適切な非機能的文の後ろに実の制御フローを隠すこと；対応する高レベル言語コンストラクトが全く存在しないオブジェクトコードレベルでコードシーケンスを導入すること；及び実の制御フローの抽象化を除去するか又はスプリアスなものを導入することが含まれる。

第１の分類（制御フロー）を考慮すると、サイクロマティック及びネスティング複雑性メトリックは、一片のコードの感知された複雑性とそれが含む述語の数の間には強い相関関係があること示唆している。不明瞭述語は、プログラム内に新しい述語を導入する変換の構築を可能にする。

図１１ａを参照すると、不明瞭述語P^Tが、$S = S_1 \cdots S_n$である基本ブロックＳ内に挿入されている。こうしてＳは半分に分割される。P^T述語は、つねに「真」まで評価することになるため、不適切なコードである。図１１ｂでは、Ｓは再び２つの半分に分割され、これらの半分は２つの異なる混乱化済みバージョンS^a及びS^bへと変換される。従ってリ

バースエンジニアにとって、S^aとS^bが同じ機能を果たすことは明白ではなくなる。図１１ｃは、図１１ｂと類似しているが、S^b内にバグが導入されている。P^T述語はつねに、コードS^aの正しいバージョンを選択する。

もう１つのタイプの混乱化変換は、データ変換である。データ変換の一例は、コードの複雑性を増大させるためにアレイを逆構造解釈することである。１つのアレイは、複数のサブアレイに分割でき、２つ以上のアレイは単一のアレイに併合でき、或いは又アレイのディメンションを増大（平坦化）又は減少（フォールディング）させることもできる。図２４は、一定数のアレイ変換例を示している。文（１－２）では、アレイＡが２つのサブアレイＡ１及びＡ２内に分割されている。Ａ１は、偶数の指標をもつ要素を含み、Ａ２は奇数の指標をもつ要素を含む。文（３－４）は、２つの整数アレイＢ及びＣが、１つのアレイＢＣを生み出すべくいかにインタリーブされうるかを例示している。Ｂ及びＣからの要素は、変換済みアレイ全体にわたり均等に拡散される。文（６－７）は、アレイＤのアレイＤ１へのフォールディングを例示している。かかる変換は、従来欠如していたデータ構造を導入するか又は既存のデータ構造を除去する。こうして、例えば、２次元アレイを宣言する上でプログラマは通常、選ばれた構造が対応するデータ上にマッピングする状態で、１つの目的のためにそれを行なうことから、プログラムのあいまいさが大幅に増大する可能性がある。そのアレイが１－ｄ構造にフォールドされたならば、リバースエンジニアは、貴重なプラグマティック情報を奪われることになるだろう。

もう１つの混乱化変換例は、予防変換である。制御又はデータ変換とは対照的に、予防変換の主たる最終目的は、人間の読み手にとってプログラムをあいまいにすることではなく、既知の自動的混乱解除技術をよりむずかしくするか又は現行の混乱解除器又はデコンパイラ内で既知の問題を開発利用することにある。このような変換は、それぞれ固有の及び目標の変換として知られている。固有予防変換の一例は、for-loopをrun backwardに再オーダすることである。このような再オーダリングは、ループが、ループ支持型データ従属性を全くもたない場合に可能である。混乱解除は同じ分析を行ないループを順方向実行に再オーダすることができる。しかしながら、逆転されたループに、偽りのデータ従属性が付加された場合、ループの識別及びその再オーダリングは防止されることになる。

混乱化変換のさらなる特定的例について、以下の好ましい実施形態の詳細な説明の部分で論述する。

好ましい実施形態の詳細な説明

オリジナルソースコード内に存在する情報の大部分又は全てを保持する形態でソフトウェアを分散させることが増々一般的になってきている。１つの重要な例が、Javaバイトコードである。かかるコードはデコンパイルが容易であるため、悪意あるリバースエンジニアリングの攻撃の危険性を増大させる。

従って、本発明の１実施形態に従って、ソフトウエアセキュリティの技術的保護のための複数の技術が提供されている。好ましい実施形態の詳細な説明において我々は、自動コード混乱化が、リバースエンジニアリングを防止するための現在最も実現性ある方法であるということを立証して行くつもりである。次に我々は、プログラムを、理解及びリバースエンジニアリングすることがさらに困難な等価物へと変換する混乱化ツールであるコード混乱化器の設計について記述する。

混乱化器は、数多くの場合においてコンパイラ最適化プログラムが使用するものに類似しているコード変換の適用に基づいている。数多くのかかる変換について記述し、それらを分類し、その効力（例えばどの程度まで人間の読み手が当惑させられるか）、弾力性（例えば自動的混乱解除の攻撃にどれほど耐えられるか）及びコスト（例えば、そのアプリケーションに対し、どれほどの性能オーバヘッドが付加されるか）に関しそれらを評価する。

最後に、さまざまな混乱解除技術（例えばプログラムスプライシング）及びそれらに対し混乱化器が利用できると考えられる対策について記述する。

１．序論

充分な時間、努力及び決意が与えられるのであれば、有能なプログラマはつねにどんなア

プリケーションにでもリバースエンジニアリングすることができるだろう。アプリケーションに対する物理的アクセスを獲得したリバースエンジニアは、（逆アセンブラ又はデコンパイラを用いて）それをデコンパイルし、次にそのデータ構造及び制御フローを分析することができる。これは手動でもでき、又プログラムスライサといったようなリバースエンジニアリングツールを用いて行なうこともできる。
リバースエンジニアリングは新しい問題ではない。しかしながら、大部分のプログラムが大型でモノリシックでかつストリップの状態で出荷されるネーティブコードであったことから、（決して不可能ではないものの）リバースエンジニアリングは難しく、近年になるまで、ソフトウェアデベロッパはリバースエンジニアリングにさほど注意を払ってこなかった、と言う問題がある。
しかしながらこの状況は、デコンパイル及びリバースエンジニアリングをするのが容易な形態でソフトウェアを配布することが増々一般的になるにつれて、変化している。重要な例としては、Javaバイトコード及びアーキテクチャニュートラル分散フォーマット（ＡＮＤＦ）がある。Javaアプリケーションは特に、ソフトウエアデベロッパにとっての問題を提起している。これらは、オリジナルJavaソース情報を事実上全て保持するハードウェア独立の仮想計算機コードである、Javaクラスファイルとして、インタネット上で配布される。従って、これらのクラスファイルはデコンパイルが容易である。しかも、計算の多くが標準ライブラリ内で行なわれることから、Javaプログラムは往々にしてサイズが小さく、従って比較的リバースエンジニアリングしやすい。
Javaデベロッパの主たる関心事は、アプリケーション全ての徹底的な再エンジニアリングではない。このような行動は、明らかに著作権法〔２９〕に違反し訴訟で争うことができることから、比較的価値のないことである。むしろ、デベロッパが最も恐れているのは、競合相手が、そのアプリケーションから所有権主張可能なアルゴリズム及びデータ構造を抽出してそれを自社のプログラム内に取込むことができるようになる、という予想である。これは競合相手に商業的な有効性（開発時間及びコストを削減することによる）を与えるばかりでなく、検出及び法的追求が困難なことでもある。この最後の点は、法律に関する無限の予算をもつ強力な企業〔２２〕に対して、長期間にわたる法律上の戦闘を行なう経済力をもたないであろう小規模デベロッパに、特にあてはまることである。
ソフトウェアの法的保護又はセキュリティを提供するためのさまざまな形態の保護の概要が図２に提供されている。図２は、（ａ）悪意あるリバースエンジニアリングに対する保護の種類、（ｂ）混乱化変換の質、（ｃ）混乱化変換により目標とされている情報、（ｄ）レイアウト混乱、（ｅ）データ混乱、（ｆ）制御混乱、及び（ｇ）予防混乱の分類を提供している。
ソフトウエアデベロッパが利用可能である知的財産の技術的保護のさまざまな形態について以下で議論する。この議論はJavaクラスファイルとしてインタネット上で配布されるJavaプログラムに限られるものであるが、結果の大部分は、当業者には明らかであるように、その他の言語及びアーキテクチャニュートラルフォーマットにもあてはまるものである。移動コードの保護に対する唯一の合理的アプローチは、コード混乱化であることを立証していく。さらに我々は、いくつかの混乱化変換を提示し、これらを有効性及び効率に応じて分類し、いかにしてそれらを自動的混乱化ツール内で使用することができるようにするかを示す。
好ましい実施形態の詳細な説明の残りの部分は、以下のように構成される。第２節で、ソフトウェアの盗難に対するさまざまな形態の技術的保護の概要を示し、コード混乱化が現在最も経済的な予防を提供していることを立証する。第３節では、現在構築中であるJavaのためのコード混乱化器であるKavaの設計の簡単な概要を示す。第４節及び５節は、異なるタイプの混乱化変換を分類し評価するのに使用される基準について記述している。第６節、７節、８節及び９節は、混乱化変換のカタログを提示している。第１０節で、より詳細な混乱化アルゴリズムを示している。第１１節では、結果のまとめ及びコード混乱化の将来の方向についての議論で締めくくっている。

２．知的財産の保護

以下の筋書きを考える。アリスは、インタネット上で彼女のアプリケーションをユーザが恐らくは有料で利用できるようにすることを望んでいる、小規模なソフトウエアデベロッパである。ボブは、自分がアリスのアプリケーションのキーアルゴリズム及びデータ構造にアクセスできた場合に、アリスに対して商業的な優位性を得ることができると考えているライバルのデベロッパである。

これは、２人の敵対者、すなわち自らのコードを攻撃から保護しようとするソフトウエアデベロッパ（アリス）及び、アプリケーションを分析しそれを容易に読取り理解できる形態に変換することを仕事とするリバースエンジニア（ボブ）、の間の２プレイヤーゲームと考えることができる。ここで、ボブがアプリケーションをアリスのオリジナルソースに幾分か近いものに変換することは不要である、という点に留意されたい。必要なのは、リバースエンジニアリングされたコードがボブ及び彼のプロクフマにとって理解可能なものである、ということだけである。同様に、アリスはボブから自らのアプリケーション全体を保護する必要すらない可能性がある、ということにも留意されたい。これは恐らく、大部分が、競合相手にとって実際関心の的でない「バタ付きパンコード」から成る。

アリスは、上述の図２ａに示されているような法的又は技術的保護のいずれかを用いて、ボブの攻撃から自らのコードを保護することができる。著作権法はソフトウエア工作物を確かに網羅しているものの、アリスのような小さい会社がより大きくより力のある競合相手に法律を遵守させることは、経済的な現実から困難なことである。より魅力的な解決法は、アリスが、リバースエンジニアリングを技術的に極めてむずかしいものとし、リバースエンジニアリングを不可能とし又は少なくとも経済的に実現するのがほとんど不可能となるようにすることによって、自らのコードを保護することにある。技術的保護におけるいくつかの初期の試みがGoslerによって記載されている。（James R. Gosler, Software protection: Myth or reality? In CRYPTO ’85 …Advances in Cryptology, pages 140-157, 1985年8月）。

最も安全なアプローチは、アリスが自らのアプリケーションを全く販売せず、むしろそのサービスを売ることである。換言すると、ユーザはアプリケーション自体にアクセスできることは決してなく、むしろ毎回少額の電子通貨を支払って図３ａに示されているように、遠隔でプログラムを走らせるためにアリスのサイトに接続する。アリスにとっての利点は、ボブがそのアプリケーションに対する物理的アクセスを獲得することは決してなく、従ってそれをリバースエンジニアリングできないということにある。そのマイナス面は当然のことながら、ネットワーク帯域幅及び待ち時間に関する限界のため、アプリケーションが、ユーザのサイト上で局所的に実行された場合よりもはるかに悪い性能を示す可能性があるということにある。部分的解決法は、アプリケーションを２つの部分、すなわち図３ｂに示されているようにユーザのサイト上で局所的に走らせる公開部分及び遠隔で走らせる（アリスが保護したいアルゴリズムを含む）私用部分に分割することである。

もう１つのアプローチは、自らのコードを例えば図４ａに示されているように、ユーザに送る前にアリスがそのコードを暗号化することであろう。残念なことにこれは、解読／実行プロセス全体がハードウェア内で行なわれる場合にのみ機能する。このようなシステムは、Herzberg（Amir Herzberg and Shlomit S. Pinter.Public protection of software. ACM Transactions on Computer Systems, 5（4）: 371-393, 1987年11月）及びWilhelm（Uwe G. Wilhelm. Cryptographically protected objects. http://lsewww. epfl. ch/-wilhelm/CryPO. html. １９９７）に記述されている。コードが仮想計算機インタプリタによりソフトウェア内で実行される場合（Java bytecodesが最も頻繁にそうであるように）、ボブが解読済みコードを傍受しデコンパイルすることはつねに可能となる。

Java™プログラミング言語は、主としてそのアーキテクチャニュートラルバイトコードのため人気を博した。これは明らかに移動コードを容易にするものの、ネーティブコードに比べ１ケタ分性能を低下させる。予想できた通りに、このことは、Javaバイトコードを実行中にネーティブコードに翻訳するジャストインタイムコンパイラの開発を導いた。アリスは、全ての一般的なアーキテクチャについて自らのアプリケーションのネーティブコードバージョンを作成するべくこのような翻訳プログラムを使用することができた。アプリ

ケーションをダウンロードするとき、ユーザのサイトは、それが走っているアーキテクチャ／オペレーティングシステムの組合せを識別しなくてはならず、例えば図４ｂに示されているように、対応するバージョンが伝送されることになるだろう。ネーティブコードにアクセスできるだけでは、ボブのタスクは、不可能ではないものの、さらにむずかしくなる。ネーティブコードを伝送する上での複雑性はさらに増す。問題は、実行前にバイトコード確認を受けるJavaバイトコードとは異なり、ネーティブコードはユーザの機械上で完全に安全に走行しえないということである。アリスが共同体の信頼されたメンバーである場合、ユーザは、アプリケーションがユーザの側で何も有害なことはしないという彼女の保証を受諾することができる。誰れもアプリケーションを汚染しようとしていないことを確かめるためには、アリスは、コードが自らの書込んだオリジナルのコードであることをユーザに立証するべく、伝送中のコードにデジタル署名しなければならないだろう。

我々が考慮する最後のアプローチは、例えば図５に示されているようなコード混乱化である。基本的な考え方は、アリスが、アプリケーションをオリジナルと機能的には同一であるもののボブにとってはるかに理解し難いものであるアプリケーションに変換するプログラムである混乱化器を通して、自らのアプリケーションを走行させるというものである。我々は、混乱化が、それに値する注目を今後受けるはずであるソフトウエア取引秘密の保護のための実現可能な技術であると信じている。

サーバ側実行とは異なり、コード混乱化は、悪意あるリバースエンジニアリング努力から１つのアプリケーションを完全に保護することは決してできない。充分な時間と決意を与えられたボブは、その重要なアルゴリズム及びデータ構造を検索するためアリスのアプリケーションを吟味することが常に可能である。この努力を助けるべく、ボブは、混乱化変換を取り消すことを試みる自動混乱化解除プログラムを通して、混乱化済みコードを走行させようとするかもしれない。

従って、混乱化器がアプリケーションに付加するリバースエンジニアリングからのセキュリティレベルは、例えば、（ａ）混乱化器によって利用される変換の精巧化、（ｂ）利用可能な混乱化解除アルゴリズムのパワー及び（ｃ）混乱解除器が利用できる資源（時間及び空間）の量によって左右される。理想的には、暗号化（大きい素数を発見することが容易）及び解読（大きい数をファクター化するのが困難）のコストの劇的な差が存在する現在の公開かぎ暗号方式における状況を模倣したいと考える。実際そこには、以下で議論するようにポリノミナルタイム（多項式的時間）において応用できるものの、混乱化解除するのにイクスポネンシャルタイム（指数的時間）を必要とする混乱化変換が存在することになるだろう。

３．Java混乱化器の設計

図６は、Java混乱化器であるKavaのアーキテクチャを示す。ツールに対する主要入力は、Javaクラスファイルセット及びユーザが要求する混乱化レベルである。ユーザは任意には、Javaプロファイリングツールによって生成されるとおり、プロファイリングデータのファイルを提供することができる。この情報は、アプリケーションのうち頻繁に実行される部分が非常に高価な変換によって混乱化されないことを確認するべく混乱化器を案内するのに使用可能である。ツールに対する入力は、Javaクラスファイルの１セットとして与えられるJavaアプリケーションである。ユーザは同様に、必要とされる混乱化レベル（例えば効力）及び混乱化器がアプリケーションに付加することを許されている最大実行時間/空間ペナルティ（コスト）を選択する。Kavaは、直接又は間接的に参照指示されたあらゆるライブラリファイルと共にクラスファイルを読みとりパージングする。完全に継承ツリーが、全てのシンボルについてのタイプ情報を示すシンボルテーブル及び全てのメソッドについての制御フローグラフと共に構築される。

Kavaは、以下で記述する大きなコード変換プールを含んでいる。しかしながら、これらが適用可能となる前に、前処理用パスが、１つの実施形態に従って、アプリケーションについてのさまざまなタイプの情報を収集する。一部の種類の情報は、手順間データフロー分析及びデータ従属性分析といった標準コンパイラ技術を用いて集めることができ、又ユーザによって提供されうるものも、又専門化された技術を用いて集められるものもある。例

えば、プラグマティック分析は、どんな種類の言語コンストラクト及びプログラミングイディオムがそれを含んでいるかをみるためアプリケーションを分析する。
前処理用パスの間に集められた情報は、適切なコード変換を選択し適用するために用いられる。アプリケーション内のあらゆるタイプの言語コンストラクトが、混乱化の対象でありうる。例えば、クラスを分割又は併合することができ、メソッドを変更又は作成することができ、又新しい制御及びデータ構造を作成すること及びオリジナルのものを修正することが可能である。アプリケーションに付加される新しいコンストラクトは、前処理パス中に集められたプラグマティック情報に基づいて、ソースアプリケーション内のものにできるかぎり類似したものとなるように選択することができる。
変換プロセスは、必要とされる効力が達成されたか又は最大コストを上回ってしまうまで反復される。ツールの出力は、機能的にオリジナルのものと等価である、Javaクラスファイルセットとして通常与えられる新しいアプリケーションである。ツールは同様に、変換がそれについて適用された情報、及び混乱化済みコードがオリジナルコードといかに関係しているかの情報が注釈として付けられたJavaソースファイルを生成することもできるだろう。注釈のついたソースは、デバッキングのために有用となる。
４．混乱化変換の分類
この好ましい実施形態の詳細な説明の残りの部分で、さまざまな混乱化変換について記述し、それを分類し評価する。まず混乱化変換の概念を形成化することから始める。
定義１（混乱化変換）
Ｐ-^{T}-＞Ｐ’を法的混乱化変換とし、ここにおいて、以下の条件が保たれなくてはならない。
－　Ｐが終結できないか又はエラー条件で終結した場合、Ｐ’は終結してもしなくてもよい。
－　そうでない場合、Ｐ’は終結し、同じ出力をＰとして生成しなければならない。
可観測性挙動は、あいまいに「ユーザが経験する通りの挙動」として定義される。これはすなわち、ユーザがその副作用を経験しないかぎり、Ｐ’は、Ｐがもたない副作用（例えばファイルの作成又はインタネット上でのメッセージ送付）を持ちうる、ということを意味する。我々はＰ及びＰ’が同じように効率のよいものであることを要求していないという点に留意されたい。実際、我々の変換のうちの多くのものの結果として、Ｐ’はＰよりも緩慢になったり、Ｐよりも多くのメモリを使用することになる。
混乱化技術の異なるクラス間での主な分割ラインが図２ｃに示されている。まず第１に、それが目標とする情報の種類に従って混乱化変換を分類する。いくつかの単純な変換は、ソースコードフォーマティングといったアプリケーションの語い構造（レイアウト）及び変数の名前を目標とする。１つの実施形態においては、興味の対象であるより精巧な変換は、アプリケーションによって用いられるデータ構造又はその制御フローのいずれかを目標とする。
第２に、それが目標とされた情報について実行するオペレーションの種類に応じて変換を分類する。図２ｄ～２ｇを見ればわかるように、対照又はデータの集合を操作する複数の変換が存在する。かかる変換は標準的にプログラマによって作成された抽象化を分解するか又は、関係のないデータ又は制御を合わせて束にすることによって新しい偽りの抽象を構築する。
同様にして、いくつかの変換は、データ又は制御のオーダリングに影響を及ぼす。数多くの場合において、２つの項目が宣言されるか又は２つの計算が行なわれる順番は、プログラムの可観測性挙動に対しいかなる効果ももたない。しかしながら、プログラムを書込んだプログラマならびにリバースエンジニアにとって、選択された順序で埋め込まれたはるかに有用な情報が存在しうる。空間的又は時間的に２つの項目又は事象が近ければ近いほど、それらがいずれかの形で関係をもつ確率は高くなる。オーダリング変換は、宣言又は計算の順序をランダム化することによりこれを調査しようとする。
５．混乱化変換の評価
いずれかの混乱化変換の設計を試みる前に、かかる変換の質を評価することができなくて

はならない。この節では、複数の基準すなわち、それらがどれほどのあいまいさをプログラムに付加するか（例えば効力）、混乱化解除器にとってそれらがいかに破壊し難いものであるか（例えば弾力性）そして、混乱化済みアプリケーションに対しそれらがどれほどの計算オーバヘッド（例えば、コスト）を付加するかという基準に従って変換を分類することを試みる。

５．１　効力の尺度

まず最初に、プログラムＰ’にとって、プログラムＰよりもさらにあいまい（又は複雑又は読取り不能）であるということが何を意味するかを定義づけする。このような全てのメトリックは、定義上、それが人間の認識能力に（一部）基づかなくてはならないことから比較的不確かなものでありうる。

幸いなことに、ソフトウエアエンジニアリングのソフトウエア複雑性メトリック分岐における多くの研究を利用することができる。この分野においては、メトリックは、読取り可能で、信頼性が高く維持できるソフトウエアの構築を助けることを意図して設計される。メトリックは往々にしてソースコードのさまざまなテクチュア特性を計数しこれらの計数値を複雑性の尺度へと組合わせることに基づいている。これまでに提案されてきた公式のいくつかは、実際のプログラムの実験的研究から誘導されてきたが、その他のものは純粋に投機的なものであった。

メトリックの文献中に見られる詳細な複雑性の公式は、「プログラムＰ及びＰ’が、Ｐ’がＰに比べより多くの特性ｑを含んでいるという点を除いて同一である場合、Ｐ’はＰよりもさらに複雑である」といったような一般的文を誘導するのに使用することができる。このような文が与えられた場合、我々は、これがそのあいまいさを増大する可能性が高いことを知りながら、プログラムに対しより多くのｑ－特性を付加する変換を構築することを試みることができる。

図７は、Ｅ（Ｘ）がソフトウェア構成要素Ｘの複雑性であり、Ｆが関数又は方法であり、Ｃがクラスであり、Ｐがプログラムである、より評判の良い複雑性尺度のいくつかを作表したテーブルである。ソフトウエア構築プロジェクトにおいて使用された場合、標準的な最終目的はこれらの尺度を最小にすることである。これとは逆に、プログラムを混乱化する場合、我々は一般的に、尺度を最大にしたいと考える。

複雑性メトリックスにより我々は効力という概念を形式化することができ、これは以下で変換の有用性の尺度として使用されることになる。非公式には、１つの変換は、それがアリスのオリジナルコードの意図を隠すことによってボブを混乱させる優れた働きをする場合に効力がある。換言すると、変換の効力は、オリジナルコードに比べ（人間にとって）混乱化済みコードがどれほど理解し難いかを測定する。これは、以Ｆの定義において形式化される：

定義２（変換効力）　Ｔを挙動保存変換とし、$P \xrightarrow{T} P'$がソースプログラムを目標プログラムＰ’に変換するようにする。Ｅ（Ｐ）を、図７のメトリックの１つにより定義される通りのＰの複雑性とする。

T_{pot}（Ｐ）すなわちプログラムＰに対するＴの効力は、ＴがＰの複雑性を変更する程度の尺度である。これは、

$$T_{pot}(P) \stackrel{def}{=} E(P')/E(P) - 1$$

として定義される。Ｔは、T_{pot}（Ｐ）＞０である場合、効力ある混乱化変換である。

この議論においては、３点目盛（低、中、高）上で効力が測定される。

テーブル１中の観察事項は、我々が変換Ｔのいくつかの望ましい特性をリストアップすることを可能にする。Ｔが効力ある混乱化変換であるためには、それが以下のことを行なうことが必要である。

－　プログラムサイズ（u_1）全体を増大させ、新しいクラス及び方法（u^a_7）を導入する。

－　新しい述語（u_2）を導入し条件付き及びルーピングコンストラクトのネスティングレベル（u_3）を増大させる。

－　メソッド引き数の数（u_5）及びクラス間インスタンス変数従属性（u^d_7）を増大さ

せる。
－　継承ツリーの高さ（$u^{b,c}{}_{7}$）を増大させる。
－　長範囲変数従属性（u_4）を増大させる。
５．２　弾力性の尺度
一見して、T_{pot}（Ｐ）を増大させることがトリビアルであると思われる。例えばu_2メトリックを増大させるために、我々がなすべきことはＰに対しいくつかの任意のif文を付加することだけである：

```
main() {                    main()  {
S1;                         S1;
S2;     = T = >             if (5==2) S1;
                                S2;}
                            if (1>2) S2;
                          }
```

残念なことに、このような変換は、単純な自動技術によって容易に取り消されうることから、事実上役に立たない。従って、自動混乱解除器の攻撃下でどれほど変換が耐えられるかを測定する弾力性の概念を導入することが必要である。例えば、変換Ｔの弾力性は、次の２つの尺度の組合せとして見ることができる：すなわち
プログラマの努力：Ｔの効力を有効に低減させることができる自動混乱化解除器を構築するのに必要とされる時間の量：及び
混乱解除器の努力：かかる自動混乱解除がＴの効力を有効に低減させるのに必要とされる時間及び空間。
弾力性と効力を区別することが重要である。変換は、それが人間の読み手を混乱させようとする場合には効力があるが、自動混乱化解除器を混乱させる場合には弾力性がある。
弾力性は、図８ａに示されているように、トリビアルからワンウェイまでの目盛上で測定される。ワンウェイ変換は、それらが決して取り消され得ないという意味で特殊である。これは標準的に、これらの変換が人間のプログラマにとっては有用であったがプログラムを正しく実行するためには必要でないプログラムからの情報を除去するからである。例としては、フォーマティングを除去し、変数名をスクランブルする変換が含まれる。
その他の変換は、標準的には、その可観測性挙動を変更しないが、人間の読み手に対する「情報負荷」を増大させるような役に立たない情報をプログラムに付加する。これらの変換は、変動する困難度で取り消しが可能である。
図８ｂは、混乱化解除器の努力がポリノミアルタイム又はイクスポネンシャルタイムのいずれかとして分類されることを示す。プログラマの努力、つまり変換Ｔの混乱化解除を自動化するのに必要とされる作業は、Ｔの範囲の関数として測定される。これは、プログラム全体に影響を及ぼしうるものに対してよりも手順のうちの小さな部分のみに影響を及ぼす混乱化変換に対する対策を構築する方が容易であるという直観力に基づいている。
変換の範囲は、コード最適化理論から借りた用語を用いて定義される：すなわち、Ｔは、それが制御フローグラフ（ＣＦＧ）の単一基本ブロックに影響を及ぼす場合、局所変換であり、ＣＦＧ全体に影響を及ぼす場合大域変換であり、手順間の情報フローに影響を及ぼす場合手順間変換であり、それが独立して実行する制御スレッド間の相互作用に影響を及ぼす場合プロセス間変換である。
定義３（変換弾力性）　Ｔを挙動保存変換とし、Ｐ＝Ｔ＝＞Ｐ’がソースプログラムＰを目標プログラムＰ’に変換するようにする。T_{res}（Ｐ）は、プログラムＰに対するＴの弾力性である。T_{res}（Ｐ）は、ＰがＰ’から再構築され得ないようにＰから情報が除去される場合、ワンウェイである。そうでなければ、
$T_{res}{}^{def}$＝弾力性（$T_{DEobfuscator\ effort}$，$T_{programmer\ effort}$）であり、ここで弾力

性は、図８ｂ中でマトリクス内に定義された関数である。

５．３　実行コストの尺度

図２ｂでは、効力と弾力性が、１つの変換を記述する３つの構成要素のうちの２つであることがわかる。第３の構成要素すなわち変換のコストは、変換が混乱化済みアプリケーションに対して招来する実行時間又は空間ペナルティである。我々は、コストを４点目盛（無料、安価、高価、法外）上で分類し、このうちの各評点について以下で定義する。

定義５（変換コスト）　Ｔを挙動保存変化とし、T_{cost}（Ｐ）∈｛法外、高価、安価、無料｝となるものとし、Ｐ’の実行がＰよりも０（１）個多い資源を必要とする場合T_{cost}（Ｐ）＝無料；Ｐ’の実行がＰよりも０（ｎ）個多い資源を必要とする場合、T_{cost}（Ｐ）＝安価；Ｐ＞１でＰ’の実行がＰよりも０（n^P）個多い資源を必要とする場合、T_{cost}（Ｐ）＝高価；又そうでなければ、T_{cost}（Ｐ）＝法外（すなわちＰ’の実行がＰよりも指数的に大きい資源を必要とする）とする。

交換に付随する実際のコストが、その適用環境により左右されることに留意すべきである。例えば、プログラムの最上レベルに挿入された単純な割当て文ａ＝５は、恒常なオーバヘッドしかこうむらない。内部ループ内部に挿入された同じ文は、実質的にさらに高いコストを有することになる。他の指摘のないかぎり、我々につねに、それがソースプログラムの最も外側のネスティングレベルで適用されたかのように変換のコストを提供する。

５．４　質の尺度

ここで、混乱化変換の質の正式の定義を示すことができる：

定義６（変換の質）

変換Ｔの質T_{qual}（Ｐ）は、Ｔの効力、弾力性及びコストの組合せとして定義される：T_{qual}（Ｐ）＝（T_{pot}（Ｐ），T_{res}（Ｐ），T_{cost}（Ｐ））。

５．５　レイアウト変換

新しい変換について調査する前に、例えばCremaといった現行のJava混乱化器に典型的であるトリビアルレイアウト変換について簡単に見ていく。（Hans Peter Van Vliet. Crema……Java　混乱化器。http://web.inter.nl.net/users/H.P.van.Vliet/crema,html, １９９６年１月）。第１の変換は、Javaクラスファイル内で時として利用可能なソースコードフォーマティング情報を除去する。これは、オリジナルフォーマッティングがひとたび過ぎ去るとそれを回復できないことから、ワンウェイ変換である；フォーマッティングにおいてきわめてわずかな意味論的内容しか存在せず、その情報が除去された時点で大きな混乱は全く導かれないことから、これは低効力の変換である；最後に、これは、アプリケーションの空間及び時間的複雑性に影響が及ぼされないことから、無料の変換である。

識別子名のスクランブリングも同様にワンウェイでかつ無料の変換である。しかしながら、それは、識別子がプラグマティック情報を多く含んでいることから、フォーマッティング除去よりもはるかに高い効力をもつ。

６．制御変換

本節及び以下の数節では、混乱化変換のカタログを紹介する。そのいくつかは、コンパイラ最適化及びソフトウエアリエンジニアリングといった他の分野で使用された周知の変換から誘導されたものであり、他のものは本発明の１実施形態に従った混乱化のみを目的として開発されたものである。

本節では、ソースアプリケーションの制御フローをあいまいにしようとする変換について論述する。図２ｆに示されているように、我々はこれらの変換を、制御の流れの集合、オーダリング又は計算に影響を与えるものとして分類する。制御集合変換は、論理的に相互帰属計算を分割するか又は共に属さない計算を併合する。制御オーダリング変換は、計算が実施される順序をランダム化する。計算変換は新しい（冗長又はデッド）コードを挿入するか又はソースアプリケーションに対しアルゴリズム変更を行なうことができる。

制御フローを変える変換については、一定量の計算オーバヘッドが不可避であろう。アリスにとってはこれは、彼女が非常に効率の良いプログラムときわめて混乱化されたプログラムの間での選択をせまられる可能性があることを意味している。混乱化器がこのトレードオフにおいて、安価な変換と高価な変換の間での選択が行なえるようにすることによっ

て彼女を補助することができる。

６．１　不明瞭な述語

制御変更変換を設計するときの実際の課題は、それらを安価にすることだけでなく、混乱化解除器からの攻撃に対し耐性あるものにすることにもある。これを達成するため、数多くの変換は、不明瞭変数及び不明瞭述語の存在に依存している。非公式には、変数Ｖは、それが先験的に混乱化器に知られているものの混乱化解除器が演繹しがたいいくつかの特性ｑを有する場合、不明瞭である。同様にして、混乱化器には周知であるもののその成果を演繹することが混乱化解除器にとってきわめて困難である場合、その述語Ｐ（論理式）は不明瞭である。

混乱化解除器にとって解明がむずかしい不明瞭な変数及び述語を作成できることが、混乱化ツールの作成者にとって主要な挑戦であり、きわめて弾力性のある制御変換への鍵である。我々は、不明瞭な変数又は述語の弾力性（即ち混乱化解除攻撃に対するその耐性）を変換弾力性と同じ目盛上で測定する（すなわちトリビアル、弱、強、フル、ワンウェイ）。同様にして我々は、不明瞭なコンストラクトの付加コストを変換コストと同じ目盛（すなわち、無料、安価、高価、法外）上で測定する。

定義７（不明瞭コンストラクト）　１つの変数Ｖは、それが、混乱化時点で知られている点ｐにおける特性ｑを有する場合、プログラム中の点ｐにおいて不明瞭である。ｐが文脈から明白である場合、我々はこれをV_q^p又はV_qと書く。述語Ｐは、その成果が混乱化時点で知られている場合、ｐにおいて不明瞭である。我々は、Ｐがｐにおいてつねに、偽（真）に評価する場合$P^F{}_p$（$P^T{}_p$）と書き、Ｐが時に真、時に偽に評価する場合、$P^?{}_p$と書く。ここでも又、ｐは、文脈から明白である場合省略されることになる。図９は、異なるタイプの不明瞭述語を示す。実線は、時としてとられ得る経路を表わし、破線は、決してとられることのない経路を示す。

以下では、単純な不明瞭コンストラクトのいくつかの例を示す。これらは、混乱化器にとって構築しやすく、混乱化解除器にとって同様に解読しやすいものである。第８節は、はるかに高い弾力性をもつ不明瞭コンストラクトの例を提供する。

６．１．１　トリビアル及び弱不明瞭コンストラクト

不明瞭コンストラクトは、混乱化解除器が統計的局所分析によりそれを解読できる（すなわちその値を演繹できる）場合、トリビアルである。分析は、それが制御フローグラフの単一基本ブロックに制限される場合、局所的である。図１０ａ及び１０ｂは、（ａ）トリビアル不明瞭コンストラクト及び（b）弱不明瞭コンストラクトの例を提供している。

我々は同様に、１つの不明瞭変数が呼出しから単純な良く理解されている意味論を用いてライブラリ関数へと計算される場合、この変数をトリビアルであるとみなす。標準的なライブラリクラスセットを支持するのに全ての実施を必要とする言語であるJava™のような言語については、かかる不明瞭変数は容易に構築される。単純な例は、ランダム（ａ，ｂ）が、ａ・・・ｂの範囲内の１つの整数を戻すライブラリ関数である$\mathrm{int}\,V^S$〔１，５〕＝ランダム（１，５）である。残念なことに、かかる不明瞭変数は同様に混乱化解除が容易である。必要なのは、全ての単純なライブラリ関数の意味論を混乱化解除器設計者が作表し次に混乱化されたコード内で関数呼出しについてパターン照合することだけである。

不明瞭コンストラクトは、静的大域分析により混乱化解除器がそれを解読できる場合、弱である。分析は、それが単一の制御フローグラフに制限されている場合に大域的である。

６．２　計算変換

計算変換は、次の３つのカテゴリに入る：すなわち、実際の計算に寄与しない無関係の文の後ろに実の制御フローを隠す；対応する高レベルの言語コンストラクトが全く存在しないオブジェクトコードレベルでコードシーケンスを導入する、又は実の制御フロー抽象を除去するか又はスプリアスなものを導入する。

６．２．１　デッドコード又は無関係コードの挿入

U_2及びU_3メトリックは、一片のコードの感知された複雑性とそれが含む述語の数の間に強い相関関係が存在することを示唆している。不明瞭述語を用いると、プログラム内に新しい述語を導入する変換を考案することができる。

図１１中の基本ブロックＳ＝S_1……S_nを考慮する。図１１ａ中では、不明瞭言語P^TをＳ内に挿入して、基本的にそれを半分に分割する。P^T述語は、それがつねに真に評価することになるため、無関係コードである。図１１ｂでは、ここでも又Ｓを２つの半分に分割し、次に、この第２の半分の２つの異なる混乱化済みバージョンS^a及びS^bを作成するべく進む。S^a及びS^bは、Ｓの第２の半分に対して異なるＯＢＳ変換セットを適用することによって作成されることになる。従って、リバースエンジニアにとって、S^a及びS^bが実際に同じ機能を果たすことは直接明白ではない。我々は、ランタイムでS^a及びS^bの間での選択を行なうために述語$P^?$を使用する。

図１１ｃは、図１１ｂと類似しているが、今度は、S^b内にバグを導入する。P^T述語はつねに、コードの正しいバージョンS^aを選択する。

６．２．２　ループ条件の拡張

図１２は、終結条件をより複雑なものにすることによっていかにしてループを混乱化できるかを示している。基本的な考え方は、ループが実行する予定の回数に影響を及ぼさないP^T又はP^F述語でループ条件を拡張するということである。例えば図１２ｄで我々が付加した述語は、$X^2(X+1)^2=0 \pmod 4$であることからつねに真に評価される。

６．２．３　可約から非可約フローグラフへの変換

往々にして、プログラミング言語は、言語自体よりもさらに表現力のあるネーティブ又は仮想計算機コードにコンパイルされる。これがあてはまる場合、こうして我々は言語分割変換を考案できるようになる。変換が言語分割変換であるのは、それがいずれのソース言語コンストラクトとも直接的対応性を全くもたない仮想計算機（又はネーティブコード）命令シーケンスを導入する場合である。かかる命令シーケンスと直面した場合、混乱化解除器は、等価（ただし重畳された）ソース言語プログラムを合成することを試みるか又は全て放棄しなければならなくなる。

例えば、Java™バイトコードは、ＧＯＴＯ命令を有するがJava™言語は、対応するＧＯＴＯ文を全くもたない。このことはすなわち、Java™バイトコードが任意の制御フローを表現でき、一方Java™言語は単に構造化された制御フローしか（容易に）表現できないということを意味している。標準的には、Java™プログラムから生成された制御フローグラフがつねに可約となるものの、Java™バイトコードは、非可約フローグラフを表現することができると言うことができる。

非可約フローグラフの表現は、ＧＯＴＯが無い言語ではきわめて扱いにくくなるため、我々は可約フローグラフを非可約フローグラフに転換する変換を構築する。これは、構造化されたループを、多重ヘッダーのあるループへと変えることによって行なうことができる。例えば、図１３ａでは、不明瞭述語P^FをWhileループに加えて、そのループの中央への飛越しが存在することがわかるようにする。事実、この分岐は決してとられることがない。

Java™デコンパイラが、コードを複製するものか又は外来のブール変数を含むものへと非可約フローグラフを変えなければならなくなる。代替的には、混乱化解除器が、混乱化器によって全ての非可約フローグラフが生成されたことを推測し、単に不明瞭述語を除去することが可能となる。これに対抗するため、時として図１３ｂに示された代替的変換を使用することができる。混乱化解除器がP^Fを盲目的に除去した場合、結果として得られるコードは正しくなくなる。

特に、図１３ａ及び１３ｂは、可約フローグラフを非可約フローグラフに変換するための変換を例示している。図１３ａでは、ループ本体Ｓ２を２つの部分（$S^a{}_2\triangledown$及び$S^b{}_2$）に分割し、偽りの飛越しを、$S^b{}_2$の始めに挿入する。図１３ｂでは、同じくＳ１を２つの部分$S^a{}_1$及び$S^b{}_1$に分割する。$S^b{}_1$はループ内に移動させられ、不明瞭述語P^Tが、S^bがつねにループ本体の前に実行されることを確実にしている。第２の述語Q^Fは、$S^b{}_1$が１度だけ実行されることを確実にしている。

６．２．４　ライブラリ呼出し及びプログラミングイディオムの除去

Javaで書き込まれる大部分のプログラムは、標準ライブラリに対する呼出しに大きく依存している。ライブラリ関数の意味論は周知であることから、かかる呼出しは、リバースエ

ンジニアに対して有利な手掛りを提供する可能性がある。Javaライブラリクラスに対する参照指示がつねに名前によるものでありこれらの名前を混乱化することはできないという事実によって問題は悪化する。
数多くの場合、混乱化器は、単純に標準ライブラリのその独自のバージョンを提供することによって、これに対抗することができるだろう。例えば、（ハッシュテーブル実施を使用する）Javaディクショナリクラスへの呼出しは、同一の挙動を伴うものの例えば赤－黒のツリーとして実施されたクラスに対する呼出しへと変えられ得る。この変換のコストは、実行時間についてはさほど大きくないがプログラムのサイズについては大きい。
類似の問題が、数多くのアプリケーションで頻繁に発生する共通のプログラミングイディオムであるクリシェ（又はパターン）でも発生する。経験豊かなリバースエンジニアは、見慣れないプログラムについての自からの理解を飛越し－開始するためにかかるパターンをサーチするだろう。一例として、Java™内のリンクされたリストを考慮する。Java™ライブラリは、挿入、削除及び列挙といった共通リストオペレーションを提供する標準クラスを全くもたない。その代り、大部分のJava™プログラマは、それらを次のフィールド上で合わせてリンクすることにより、特別のものとしてオブジェクトリストを構築することになる。かかるリストを通しての反復は、Java™プログラムにおいては非常に共通したパターンである。自動化されたプログラム認識の分野で発明された技術（本書に参考とし内含されているLinda Mary Wills, 自動化プログラム認識；フィージビリティの立証、人工頭脳、４５（１－２）；１１３－１７２，１９９０参照を参照のこと）が、共通パターンの識別及びさほど明白でないものとの置換のために使用可能である。例えばリンクされたリストの場合には、要素アレイ内にカーソルといったさほど共通でないもので標準リストデータ構造を表わすかもしれない。
６．２．５　テーブル解釈
最も有効な（かつ高価な）変換の１つは、テーブル解釈である。考え方は、コードの１区分（この例ではJavaバイトコード）を異なる仮想計算機コードに転換するというものである。この新しいコードはこのとき、混乱化済みアプリケーションと共に内含された仮想計算機インタプリタにより実行される。明らかに、特定のアプリケーションには、各々異なる言語を受諾し混乱化済みアプリケーションの異なる区分を実行する複数のインタプリタが含まれている可能性がある。
各々の解釈レベルについて通常１桁の減速が存在することから、この変換は、全体的ランタイムの小さな部分を構成するか又は非常に高レベルの保護を必要とするコードの区分に予約されたものであるべきである。
６．２．６　冗長オペランドの付加
ひとたびいくつかの不明瞭変数を構築したならば、算術式に冗長オペランドを付加するために代数法則を使用することができる。こうして、U_1メトリックが増大されることになる。明らかにこの技術は、数値的な制度が問題でない整数式の場合に最もうまく機能する。以下の混乱化済みの文（１’）では、値が１である不明瞭変数Ｐが使用される。文（２’）では、値が２である不明瞭部分式Ｐ/Ｑを構築する。明らかに、文（２’）に達した時につねにそれらの商が２となるかぎりにおいて、プログラム実行中に、Ｐ及びＱに異なる値をとらせることができる。
（１）　Ｘ＝Ｘ＋Ｖ；$=^{T}=>$（１’）　Ｘ－Ｘ＋Ｖ＊$Ｐ^{=1}$；
（２）　Ｚ＝Ｌ＋１；　　（２’）　Ｚ＝Ｌ＋（$Ｐ^{=2Q}$／$Ｑ^{=P/2}$）／２
６．２．７　コードの並列化
自動並列化は、マルチプロセッサ計算機上でランするアプリケーションの性能を増大させるのに用いられる重要なコンパイラ最適化である。我々がプログラムを並列化することを望む理由は、当然のことながらさまざまである。我々は、性能を増大させるためではなく、実際の制御フローをあいまいにするために並行性を増大することを望んでいる。考えられる２つのオペレーションが利用可能である：すなわち、
１．有用なタスクを全く行なわないダミープロセスを作成することができる。又
２．アプリケーションコードの逐次区分を、並行して実行する多重区分へと分割すること

ができる。
アプリケーションが単一プロセッサ計算機上でランしている場合、我々はこれらの変換が多大な実行時間ペナルティを有することを予想することができる。これは、これらの変換の弾力性が高いことから、数多くの状況において受諾できるものである。すなわち、プログラムを通しての考えられる実行経路の数が実行プロセス数と共に指数的に大きくなることから、並列プログラムの静的分析は非常にむずかしい。並列化は又、高レベルの効力をも生み出す：すなわち、リバースエンジニアは、逐次プログラムに比べ、並列プログラムがはるかに理解し難いものであることがわかるだろう。
図１４に示されているように、１コード区分は、それがデータ従属性を全く含まない場合、容易に並列化されうる。例えば、S_1及びS_2が２つのデータ独立型文である場合、これらは並行してランされ得る。Java™言語といったような明示的な並列コンストラクトを全くもたないプログラミング言語においては、プログラムはスレッド（軽量プロセス）ライブラリに対する呼出しを用いて並列化され得る。
図１５で示されているように、データ従属性を含むコード区分は、アウェイアンドアドバンスといった適切な同期化基本命令を挿入することにより、同時並行スレッドへと分割され得る（本書に参考として内含されているMichael Wolfe, 並列計算用高性能コンパイラ. Addison-Wesley, １９９６。ＩＳＢＮＯ－８０５３－２７３０－４を参照のこと）。このようなプログラムは基本的に逐次的にランしているが、制御フローは、１つのスレッドから次のスレッドへとシフトしていることになる。
６．３　集合変換
プログラマは、抽象化を導入することによってプログラミングの固有の複雑性を克服する。１つのプログラムの数多くのレベルで抽象化が存在するが、手順抽象化は最も重要なものである。このような理由から、混乱化器にとっては、手順及びメソッド呼出しをあいまいにすることが重要であり得る。以下では、メソッド及びメソッド呼出しをあいまいにすることのできるいくつかの方法、すなわちインライン処理、アウトライン処理、インタリーブ及びクローニングについて考慮する。これら全ての背後にある基本的考えは、同じである：すなわち（1）プログラマが１つの方法に集合した（恐らくはそれが論理的に相互帰属していたため）コードは分割され、プログラム全体にわたり分散されるべきであること、及び（2）相互に帰属していないと思われるコードは１つのメソッドの形に集合させられるべきであることである。
６．３．１　インライン及びアウトライン方法
インライン処理は、当然のことながら、重要なコンパイラ最適化である。これは、プログラムから手順抽象を除去することから、きわめて有用な混乱化変換でもある。インライン処理は、手順呼出しがひとたび、呼出された手順の本体と置換され、手順自体が除去された場合、コード内には抽象の痕跡が全く残らないことから、きわめて弾力性の高い変換である（これは基本的にワンウェイである）。図１６は、手順Ｐ及びＱがその呼出しサイトでいかにインライン処理され、次にコードから除去されるかを示す。
アウトライン処理（一続きの文をサブルーチンへと変えさせること）は、インライン処理に対する非常に有用な姉妹変換である。我々は、Ｑのコードの始まり及びＰのコードの終りを新しい手順Ｒへと抽出することにより偽りの手順抽象を作成する。
Java™言語といったようなオブジェクト指向目語では、インライン処理は、事実上、つねに完全にワンウェイの変換であるとはかぎらない。メソッド呼出しｍ．Ｐ（）を考えてみよう。呼出される実際の手順は、ｍのランタイムタイプによって左右されることになる。複数のメソッドを特定の呼出しサイトで呼出すことかできる場合、我々は考えられる全てのメソッドをインライン処理し（本書に参考として内含されているJeffrey Dean, オブジェクト指向言語の全プログラム最適化。ワシントン大学博士論文、１９９６を参照のこと）、ｍ）ａタイプについて分岐することにより適切なコードを選択する。従って、メソッドのインライン処理及び除去の後でさえ、混乱化済みコードはなおも、オリジナル抽象の痕跡を幾分か含んでいる可能性がある。例えば、図１７は、メソッド呼出しのインライン処理を例示している。ｍのタイプを統計的に決定できるのでないかぎり、ｍ．Ｐ（）を結

びつけることのできる考えられる全てのメソッドが呼出しサイトでインライン処理されなくてはならない。

６．３．２　インタリーブメソッド

インタリーブメソッドの検出は、重要で困難なリバースエンジニアリングタスクである。図１８は、同じクラスで宣言された２つのメソッドをどのようにインタリーブできるかを示す。考え方は、メソッドの本体及びパラメータリストを併合し余分のパラメータ（又は大域変数）を付加して個々のメソッドに対する呼出しを弁別するというものである。理想的には、メソッドは共通のコード及びパラメータの併合を可能にするべく性質が類似しているべきである。図１８の場合がこれであり、ここではＭ１及びＭ２の第１のパラメータは同じタイプをもつ。

６．３．３　クローンメソッド

サブルーチンの目的を理解しようとするとき、リバースエンジニアは当然のことながらその署名及び本体を検査することになる。しかしながら、ルーチンの挙動を理解するのに同じように重要なのは、それが呼出しされている異なる環境である。我々は、実際にはそうでないのに異なるルーチンが呼出されつつあるように見えるようにするべくメソッドの呼出しサイトを混乱化することによって、このプロセスをさらに難しくすることができる。図１９は、オリジナルコードに異なる混乱化変換セットを適用することによってメソッドの複数の異なるバージョンを作成することができる。我々は、ランタイムで異なるバージョン間で選択を行なうのにメソッドディスパッチを使用する。

メソッドクローニングは、図１１の述語挿入変換と類似しているが、ここでは、コードの異なるバーション間で選択を行なうのに不明瞭述語ではなくメソッドディスパッチを使用しようとしているという点が異なっている。

６．３．４　ループ変換

（特に）数値アプリケーションの性能を改善することを意図して数多くのループ変換が設計されてきた。広範な調査についてはBacon〔２〕を参照のこと。これらの変換のいくつかは、図７に関して、上述した複雑性メトリックも増大させることから、我々にとって有用である。図２０ａに示されているようなループブロック化は、内部ループが、キャッシュ内にフィットするように相互作用空間を分割することによりループのキャッシュ挙動を改善するために使用される。ループアンロールは、図２０ｂに示されている通り、１回又は複数回ループの本体を複製する。コンパイル時点でループ境界がわかっている場合、そのループを全体的にアンロールすることができる。図２０ｃに示されているようなループ分裂は、複合本体を伴うループを同じ反復空間を伴う複数のループへと変える。

３つの変換は全て、ソースアプリケーションの合計コードサイズ及び条件数を増大させることから、U_1及びU_2メトリックを増大させる。ループブロック化変換は同様に、余分のネスティングを導入し、従ってU_3メトリックも増大させる。

分離して適用された場合、これらの変換の弾力性はきわめて低い。混乱化解除器がアンロールされたループを再度ロールするのに多大な静的分析は必要でない。しかしながら変換が組合わされた時点で、弾力性は劇的に上昇する。例えば、図２０ｂの単純なループが与えられた場合、我々はまず最初にアンロールを適用し、次に分裂、そして最後にブロック化を適用することができる。結果として得たループをそのオリジナル形態に戻すには、混乱化解除器にとってかなりの量の分析が必要となるだろう。

６．４　オーダリング変換

プログラマは、その局所性を最大限にするように自らのソースコードを組織する傾向をもつ。考え方は、論理的に関係する２つの項目がソーステキスト内で同様に物理的に近い場合に、プログラムがさらに読取り、理解しやすくなるというものである。この種の局所性は、ソースの全てのレベルで機能する：例えば、式内の項の間、基本ブロック内の文、方法内の基本ブロック、クラス内の方法そしてファイル内のクラスの間に局所性が存在する。全ての種類の空間的局所性がＲＥに対し有用な手掛りを提供することができる。従って、可能な場合にはつねに、我々はソースアプリケーション中の任意の項目の配置をランダム化する。いくつかのタイプの項目（例えばクラス内のメソッド）について、これはトリ

ビアルである。その他のケースでは（例えば基本ブロック内の文）、どの再オーダリングが技術的に有効であるかを見極めるため、データ従属性分析（本書に参考として内含されている（David F. Bacon, Susan L. Graham, 及びOliver J. Sharp. 高性能計算のためのコンパイラ変換ACM Computing Surveys, ２６（４）：３４５－４２０, １９９４年１２月、http://www.acm.org/pubs/toc/Abstracts/０３６０－０３００/１９７４０６．html. 及びMichael Wolfe. 並列計算のための高性能コンパイラ。Addison-Wesley, １９９６. ISBN ０－８０５３－２７３０－４, を参照のこと）が行なわれる。
これらの変換は低い効力を有する（プログラムに多大なあいまいさを付加しない）が、その弾力性は高く、数多くのケースでワンウェイである。例えば、基本ブロック内の文の配置がランダム化された時点で、結果として得たコードにはもとの順序の痕跡は全く残っていないことになる。
オーダリング変換、第６，３，１節の「インライン－アウトライン」変換に対する特に有用な姉妹変換である。その変換の効力は、（1）手順Ｐ内で複数の手順呼出しをインライン処理すること、（2）Ｐ内の文の順序をランダム化すること及び、（3）Ｐの文の隣接区分をアウトライン処理することによって増強され得る。このようにして、以別複数の異なる手順の一部であった無関係な文が、偽りの手順抽象内に合わせて入れられる。
いくつかのケースでは、例えば後向きにランさせることによって、ループを再オーダリングすることも可能である。このようなループ逆転変換は、高性能コンパイラにおいて共通である。（David F. Bacon, Susan L. Graham, 及びOliver J. Sharp. 高性能計算のためのコンパイラ変換ACM Computing Surveys, ２６（４）：３４５－４２０, １９９４年１２月、http://www.acm.org/pubs/toc/Abstracts/０３６０－０３００/１９７４０６．html.）。

７．データ変換

本節では、ソースアプリケーションで使用されるデータ構造を混乱化する変換について説明する。図２ｅに示しているように、こうした変換は、データの記憶、符号化、集合、または、オーダリングを行う変換として分類される。

７．１　記憶および符号化変換

多くの場合には、プログラム中の特定のデータ項目を記憶するための「自然な」方法がある。例えば、１つの配列の各要素を繰り返すためには、反復変数として適切なサイズのローカル整変数を割り当てることが選択されるだろう。他の変数タイプが使用可能であるが、こうした他の変数タイプは自然さの点で劣るだろうし、恐らくは効率的にも劣ることだろう。
さらに、変数のタイプに基づいている特定の変数が有することが可能であるビットパターンの「自然な」解釈が存在する場合も多い。例えば、一般的に、ビットパターン「０００００００００００１１００」を記憶する１６ビット整変数が整数値「１２」を表すと仮定される。当然のことながら、これらは単なる取決めであり、他の解釈が可能である。
混乱化記憶変換は、動的データと静的データとのための不自然な記憶域クラスを選択することを試みる。同様に、符号化変換は、共通データタイプに関して不自然な符号化を選択しようとする。記憶変換と符号化変換とが組み合わせて使用される場合が多いが、場合によっては、これらの変換の各々が単独で使用されることも可能である。

７．１．１　符号化の変更

符号化変換の単純な事例として、c_1とc_2とが定数である場合に、$i_0 = c_1 * i + c_2$で整変数ｉを置き換える。効率のために、c_1を２の累乗として選択することが可能である。次の例では、$c_1 = 8$、$c_2 = 3$とすると、

```
{                          = T =>      {
int i=1;                               int i=11;
while (i < 1000)                       while (i < 8003)
     . . . A[i] . . .;                   . . A[(i-3)/8] . . .;
    i++;                                    i+=8;
}                                      }
```

当然のことながら、オーバフロー（および、浮動小数点変数の場合には、正確度）の問題に対処する必要がある。問題の変数の範囲（この範囲は、静的解析法を使用することによってまたはユーザに質問することによって決定されることが可能である）のせいでオーバフローが発生しないということ、または、より大きな変数タイプに変更可能であるということを調べることが可能である。

一方では弾力性と効力との間のトレードオフ、他方では弾力性とコストとの間のトレードオフとがあり得る。上記例の$i_0 = c_1 + i + c_2$のような単純な符号化関数は僅かな追加の実行時間しか付加しないが、一般的なコンパイラ解析法（Michael Wolfe. High performance Compilers For Parallel Computing. Addison-Wesley, 1996. ISBN 0-8053-2730-4、および、David F. Bacon, Susan L. Graham and Oliver J. Sharp. Compiler transformations for high-performance computing. ACM Computing Surveys, 26（4）: 345-420, December 1994. http://www.acm.org/pubs/toc/Abstracts/0360-0300/197406.html）を使用して混乱解除されることが可能である。

７．１．２　変数のプロモート

特殊化された記憶域クラスからより汎用のクラスに変数をプロモートする単純な記憶変換が幾つか存在する。こうした記憶変換の効力と弾力性は一般的に低いが、他の変換と組み合わせて使用される場合には、極めて有効であり得る。例えば、Javaでは、整変数が整数オブジェクトにプロモートされることが可能である。同じことが、対応する「パッケージ化された」クラスを全てが有する他のスカラ型の場合に当てはまる。Java™が不要部分の整理をサポートするので、オブジェクトがすでに参照されなくなっている時には、そのオブジェクトが自動的に除去されるだろう。ここに、その一例がある。

```
 {                              {
 int I=1;                       int i = new int(1);
 while (i < 9)   = T =>         while (i.value < 9)
      ...A[i]...;                    ...A[i.value]...;
      i++;                           i.value++;
}                              }
```

変数の寿命を変更することも可能である。最も単純なこうした変換は、独立した手続き呼出しの間で後で共有されるグローバル変数へとローカル変数を変更する。例えば、手続きＰと手続きＱの両方がローカル整変数を参照し、かつ、ＰとＱとが両方とも同時にアクティブであることが不可能である（プログラムがスレッドを含まなければ、これは、静的呼出しグラフ（static call graph）を調べることによって求められることが可能である）場合には、変数がグローバル変数にされて、これらの手続きの間で共有されることが可能である。

```
void P()  {              int C;
int i; ...I...           void P()  {
                         ...C...
                          }
void Q()      = τ =>     while (i.value<9)
int k;...k...            ...C...
}                         }
```

ＰとＱとによって参照されるグローバルデータ構造の数が増大させられるので、この変換は、u_5メトリックを増大させる。

７．１．３　変数の分割

ブール変数と制限された範囲内の他の変数とが、２つ以上の変数に分割されることが可能である。ｋ個の変数p_1，．．．，p_kに分割された変数ＶをＶ＝［p_1，．．．，p_k］と記述する。典型的には、この変換の効力がｋとともに増大するだろう。残念なことに、変換コストもｋとともに増大し、したがって、ｋを２または３に制限することが一般的である。

タイプＴの変数ＶがタイプＵの２つの変数ｐ、ｑに分割されることを可能にするためには、３つの情報断片、すなわち、（１）対応するＶの値にｐとｑの値をマップする関数Ｆ（ｐ；ｑ）、（２）対応するｐ、ｑの値にＶの値をマップする関数ｇ（Ｖ）、（３）ｐとｑとに対する演算の観点からキャストされた新たな演算（タイプＴの値に対する基本演算に対応する）が、与えられることが必要である。この節の残りの部分では、Ｖがブール変数タイプであり、ｐとｑとが小整変数であると仮定する。

図２１ａは、分割ブール変数に関して行われることが可能である表現の選択を示す。この表は、Ｖがｐとｑとに分割されている場合に、および、プログラムの何らかのポイントでｐ＝ｑ＝０またはｐ＝ｑ＝１である場合に、それが、Ｖが偽であることに相当するということを示している。同様に、ｐ＝０かつｑ＝１、または、ｐ＝１かつｑ＝０が、真に相当する。

この新たな表現で示される場合には、様々な組込みブール演算（例えば、ＡＮＤ、ＯＲ）に関する置換えが考案されなければならない。１つのアプローチは、各々の演算子に関する実行時ルックアップテーブルを提供することである。「ＡＮＤ」と「ＯＲ」とに関するテーブルが図２１ｃと図２１ｄとに別々に示されている。２つのブール変数Ｖ１＝［ｐ，ｑ］とＶ２＝［ｒ，ｓ］とが与えられていると仮定すると、$V_1 \& V_2$が、ＡＮＤ［２ｐ＋ｑ，２ｒ＋ｓ］として計算される。

図２１ｅには、３つのブール変数Ａ＝［ａ１，ａ２］、Ｂ＝［ｂ１，ｂ２］、Ｃ＝［ｃ１，ｃ２］の分割の結果が示されている。本発明者が選択した表現の興味深い側面は、同じブール式を計算するために使用可能な方法が幾つかあるということである。例えば、図２１ｅの文（３′）と文（４′）は、両方とも偽を変数に割り当てるが、互いに異って見える。同様に、文（５′）と文（６′）は互いに全く異っているが、両方ともＡ＆Ｂを計算する。

この変換の効力と弾力性とコストとの全てが、オリジナルの変数が分割される変数の個数に応じて増大する。弾力性は、実行時に符号化を選択することによってさらに増強される。言い換えれば、図２１ｂから図２１ｄまでの実行時ルックアップテーブルは、コンパイル時には構築されないが（このことが、実行時ルックアップテーブルに対して静的解析を行うことを可能にするだろう）、混乱化アプリケーションに含まれるアルゴリズムによって構築される。当然のことながら、このことが、図２１ｅでの文（６′）で行われるように、基本演算を計算するためにインラインコードを使用することを防止する。

７．１．４　静的データの手続きデータへの転換
静的データ、特に文字列は、リバースエンジニアにとって有用な実際的情報を多く含んでいる。静的ストリングを混乱化するための方法は、静的ストリングをそのストリングを生成するプログラムの形に変換することである。ＤＦＡまたはトライ走査（Trie traversal）であることが可能であるプログラムが、他のストリングも生成することが可能である。例えば、ストリング“ＡＡＡ”、“ＢＡＡＡＡ”、“ＣＣＢ”を混乱化するように構築されている図２２の関数Ｇを考察しよう。Ｇによって生成される値は、Ｇ（１）＝“ＡＡＡ”、Ｇ（２）＝“ＢＡＡＡＡ”、Ｇ（３）＝Ｇ（５）＝“ＣＣＢ”、および、（実際にはプログラムで使用されない）Ｇ（４）＝“ＸＣＢ”である。他の引数値の場合には、Ｇが終了しても終了しなくてもよい。
当然のことながら、全ての静的ストリングデータの計算を単一の関数の形に集合することは、極めて望ましくない。ソースプログラムの「通常の」制御流れの中に埋め込まれたより小さなコンポーネントの形にＧ関数が分割されている場合には、はるかに高度な効力と弾力性とが実現される。
この手法を節６．２．５のテーブル解釈変換と組み合わせることが可能であるということを指摘しておくことが重要である。その混乱化の意図は、Javaバイトコードの１つのセクションを別の仮想計算機のためのコードに変換するということである。この新たなコードが、典型的には、被混乱化プログラム内の静的ストリングデータとして記憶されるだろう。しかし、さらに高いレベルの効力と弾力性とを得るためには、上記ストリングが、上記のように、そのストリングを生成するプログラムに転換されることも可能である。
７．２　集合変換
命令型言語および機能言語とは対照的に、オブジェクト指向言語は、制御指向であるよりもデータ指向である。言い換えれば、オブジェクト指向プログラムでは、制御が、データ構造まわりに編成されるが、他の仕方では編成されない。このことは、オブジェクト指向アプリケーションのリバースエンジニアリングの重要部分は、プログラムのデータ構造の復元を試みることであることを意味する。逆に、混乱化器がこうしたデータ構造を隠蔽しようとすることが重要である。
オブジェクト指向言語の殆どでは、データの集合を行うための方法は２つだけであり、すなわち、配列の形でのデータ集合と、オブジェクトの形でのデータ集合である。次の３つのセクションでは、こうしたデータ構造が混乱化されることが可能な方法を検討する。
７．２．１　スカラ変数の併合
V_1，．．．，V_kの組合せ範囲がV_Mの精度に適合するならば、２つ以上のスカラ変数V_1，．．．，V_Kが、１つの変数V_Mの形に併合されることが可能である。例えば、２つの３２ビット整変数が、１つの６４ビット変数に併合されることが可能である。個々の変数に対する演算が、ＶＭに対する演算の形に変換されるだろう。簡単な例として、２つの３２ビット整変数Ｘ、Ｙを６４ビット変数Ｚの形へ併合することを考察する。次の併合式
$Z(X,Y)=2^{32}*Y+X$
を使用して、図２３ａの算術恒等式が得られる。図２３ｂには幾つかの簡単な例が示されている。特に、図２３は、２つの３２ビット整変数Ｘ、Ｙを１つの６４ビット変数Ｚへ併合することを示している。ＹがＺの上側３２ビットを占め、かつ、Ｘが下側３２ビットを占める。ＸまたはＹのどちらかの実際の値域がプログラムから演繹されることが可能である場合には、直感的により一層分かりにくい併合が使用されることも可能である。図２３ａは、ＸとＹによる加算と乗算とのための規則を示している。図２３ｂは、幾つかの簡単な例を示す。この例は、例えば（２′）と（３′）とを「Ｚ＋＝４７２４４６４０２６１」の形に併合する事によって、さらに混乱化されることが可能である。
変数併合の弾力性は極めて低い。ある特定の変数が実際には２つの併合変数から成るということを推定するためには、混乱解除器は、算術演算セットがその特定の変数に適用されていることを調べるだけでよい。個々の変数に対する妥当な演算のいずれにも対応することが不可能であるボーガス（ｂｏｇｕｓ）演算を導入することによって、弾力性を増大させることが可能である。図２３ｂの例では、例えば回転によって、すなわち、ｉｆ（P^F

）Ｚ＝ｒｏｔａｔｅ（Ｚ，５）によって、Ｚの２つの半分部分を併合するように見える演算を挿入することが可能である。
この変換の１つの変形は、V_1，．．．，V_Kを、次のような適切なタイプの１つの配列に併合することである。
V_A=1...k
V_1...V_k
V_1，．．．，V_kがオブジェクト参照変数である場合には、例えば、ＶＡの要素タイプが、継承階層においてV_1，．．．，V_kのタイプのいずれよりも高いレベルにある全てのクラスであることが可能である。

７．２．２　配列の再構成

配列に対して行われる演算を混乱化するために、幾つかの変換が考案されることが可能である。例えば、１つの配列を幾つかの二次配列に分割するか、２つ以上の配列を１つの配列に併合するか、１つの配列を折り畳む（ｆｏｌｄ）（次元数を増大させる）か、または、１つの配列を平坦化する（ｆｌａｔｔｅｎ）（次元数を減少させる）ことが可能である。
図２４は、配列再構成の幾つかの例を示す。文（１－２）では、配列Ａが２つの二次配列Ａ１、Ａ２の形に分割される。Ａ１が、偶数のインデックスを有するＡの要素を保持し、Ａ２が、奇数のインデックスを有する要素を保持する。
図２４の文（３－４）が、どのようにして整数配列Ｂ、Ｃが結果としての配列ＢＣの形に交互配置されることが可能であるかを示している。配列Ｂからの要素と配列Ｃからの要素とが、その結果得られる配列全体にわたって均一に分散させられている。
文（６－７）は、１次元配列Ｄが２次元配列Ｄ１の形にどのように折り畳まれることが可能であるかを示している。最後に、文（８－９）は逆変換を示している。２次元配列Ｅが１次元配列Ｅ１の形に平坦化される。
配列の分割と折畳みとがu_6データ複雑性メトリックを増大させる。一方、配列の併合と平坦化とが、このメトリックを減少させる。このことは、これらの変換が僅かなまたはネガティブな効力だけしか持たないということを示しているように見えるかも知れないが、実際には、これは誤りを生じさせやすい。問題は、幾つかのデータ構造変換の重要な側面を把握することには、図７の複雑性メトリックが役立たないということである。すなわち、こうした変換が、当初は存在しなかった構造を導入するか、または、オリジナルのプログラムから構造を取り除くことになる。このことは、プログラムの混乱化を著しく増大させることが可能である。例えば、２次元配列を宣言するプログラマは、意図的にそうするのである。選択された構造が、操作されているデータをとにかく適切にマップする。その配列が１次元構造に折り畳まれる場合には、リバースエンジニアは、貴重な実際的情報を奪われてしまっていることになるだろう。

７．２．３　継承関係の修正

Java™言語のような現在のオブジェクト指向言語では、主要なモジュール化および抽象化概念はクラスである。クラスは、データ（インスタンス変数）と制御（方法）とをカプセル化する本質的に抽象的なデータタイプである。クラスはＣ＝（Ｖ，Ｍ）と記述され、前式のＶがＣのインスタンス変数のセットであり、Ｍがその方法である。
抽象データタイプの従来の概念とは対照的に、２つのクラスC_1、C_2が、集合化（C_2がタイプC_1のインスタンス変数を有する）と継承（新たな方法とインスタンス変数とを加えることによって、C_2がC_1を拡張する）とによって構築されることが可能である。継承はC_2＝C_1ＵC'_2と記述される。C_2は、そのスーパークラスまたは親クラスであるC_1を継承すると表現される。Ｕ演算子は、C'_2で定義される新たなプロパティと親クラスとを組み合わせる関数である。Ｕの正確なセマンティクスは、個々のプログラミング言語に依存している。Javaのような言語では、一般的に、Ｕが、インスタンス変数に適用される場合には合併として解釈され、一方、方法に適用される場合にはオーバーライドと解釈される。
メトリックu_7にしたがって、クラスC_1の複雑性が、継承階層と直接の子孫の個数とにお

けるその深さ（ルートからの距離）に応じて増大する。例えば、この複雑性を増加させることが可能な方法が２つある。すなわち、図２５ａに示されているように、ある１つのクラスを分割（ファクタ）することと、図２５ｂに示されているように、新たな、にせ（bogus）の、クラスを挿入することとが可能である。

クラスファクタリングに関する問題点はその弾力性である。ファクタされたクラスを混乱解除器が簡単に併合することを阻止するものは全くない。これを防止するために、一般的には、図２５ｄに示されているように、ファクタリングと挿入とが組み合わされる。これらのタイプの変換の弾力性を増加させる別の方法は、導入されたクラス全てに関して新たなオブジェクトが生成されることを確実なものにすることである。

図２５ｃは、「偽のリファクタリング」と呼ばれる、クラス挿入の変形を示している。リファクタリングは、その構造が劣化してしまっているオブジェクト指向プログラムを再構築するための（時として自動の）手法である（本明細書に参考として採り入れられている、William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. In Stan C.Kwansny and John F. Buck, editors, Proceedings of the 21st Annual Conference on Computer Science, page 66-73, New York, NY, USA, February 1993. ACM Press. ftp://st. cs.uiuc.edu/pub/papers/refactoring/refactoring-superclasses.psを参照されたい）。リファクトリングは２ステップのプロセスである。第１に、見掛け上は別個の２つのクラスが事実上は同様の動作をするということが検出される。その次に、両方のクラスに共通する特徴が、新たな（おそらくは抽象的な）親クラスの中に移される。偽のリファクタリングは同様の操作であるが、共通の動作を持たない２つのクラスC_1、C_2に対してだけ行われる。両方のクラスが同じタイプのインスタンス係数を有する場合には、これらのクラスが新たな親クラスC_3の中に移される。C'_3の方法が、C_1とC_2からの方法のバグだらけ（ｂｕｇｇｙ）なバージョンであり得る。

７．３　オーダリング変換

セクション６．４では、（可能な場合にではあるが）計算が行われる順序をランダム化することが有用な混乱化であることを示した。同様に、ソースアプリケーションにおける宣言の順序をランダム化することが有益である。

特に、本発明においては、クラス内の方法およびインスタンス変数と方法の仮パラメタ方法との順序をランダム化する。後者の場合には、当然のことながら、対応する実際の順序が再オーダリングされなければならない。こうした変換の効力は低く、弾力性は片方向である。

多くの場合には、配列中の要素を再オーダリングすることも可能だろう。簡単に述べると、本発明では、オリジナルの配列内のｉ番目の要素を、再オーダリングされた配列の新たな位置にマップする、不明瞭性（ｏｐａｑｕｅ）符号化関数ｆ（ｉ）が提供される。

```
{                                          {
int i=1, A[1000];                          int i=1, A[1000];
while (i < 1000)          = T =>           while (i < 1000)
    ...A[i]...;                                ...A[f(i)]...;
    i++;                                       i++;
}                                          }
```

８．不明瞭値と不明瞭述語

上記のように、不明瞭な述語が、制御流れを混乱化する変換の設計における主要なビルディングブロックである。事実として、殆どの制御変換の品質が、こうした述語の品質に直接的に依存している。

セクション６．１では、トリビアルで弱い弾力性を有する単純な不明瞭述語の例を示した。これは、ローカル静的分析またはグローバル静的解析を使用して不明瞭述語が解読され

ることが可能である（自動混乱解除器がその値を発見することができる）ということを意味する。当然のことながら、一般的に、攻撃に対するはるかにより高度の抵抗性が必要とされている。理想的には、それを解読するのには（プログラムのサイズにおいて）最悪指数関数的時間を要するが、それを構築するには多項式的時間しか要さない不明瞭述語を、構築することが可能であることが望ましい。このセクションでは、２つのこうした手法を説明する。第１の手法はエイリアシングに基づいており、第２の手法は軽量プロセス（ｌｉｇｈｔｗｅｉｇｈｔ　ｐｒｏｃｅｓｓ）に基づいている。

８．１　オブジェクトとエイリアスとを使用する不明瞭構造体

エイリアシングが可能である場合には常に、手続き間静的分析が著しく複雑化される。実際に、動的割付けとループとＩＦ文を有する言語においては、正確で流れ依存形のエイリアス解析は決定不可能である。

このセクションでは、低コストでかつ自動混乱解除攻撃に対して弾力性がある不明瞭述語を構築するために、エイリアス解析の困難さが利用される。

８．２　スレッドを使用する不明瞭構造体

並列プログラムは順次プログラムに比べて静的分析を行うこと困難である。その理由は、並列プログラムのインターリービングセマンティクスである。すなわち、並列領域ＰＡＲ　S_1，S_2，．．．，S_n，ＥＮＤＰＡＲ内のｎ個の文が、ｎ！個の異った方法で実行されることが可能である。これにも係わらず、並列プログラムの幾つかの静的解析が、多項式的時間［１８］で行われることが可能であり、一方、他は、ｎ！個のインタリービング全てが考慮されることを必要とする。

Ｊａｖａでは、並列領域が、スレッドとして知られている軽量プロセスを使用して構築される。（本発明者の視点から見て）Ｊａｖａスレッドは２つの有益な特性を有する。すなわち、（１）Ｊａｖａスレッドのスケジューリングポリシーは言語仕様によっては厳密に指定されておらず、したがって、インプリメンテーションに依存することになり、（２）スレッドの実際のスケジューリングが、ユーザインタラクションによって生成される非同期イベントのような非同期イベントと、ネットワークのトラフィックとに依存することになるという特性を有する。並列領域の固有インタリービングセマンティクスと組み合わされる時には、このことは、スレッドを静的解析することが非常に困難であるということを意味する。

本発明では、解読には最悪指数関数的時間を必要とする不明瞭述語（図３２を参照されたい）を生成するために、上記の考察結果が利用される。この基本的な着想は、セクション８．２で使用されている着想と非常に類似している。すなわち、グローバルデータ構造Ｖが生成され、時たま更新されるが、不明瞭問合せが行われることが可能であるような状態に維持される。相違点は、現在実行中のスレッドによってＶが更新されるということである。

当然のことながら、Ｖは、図２６で生成された動的データ構造のような動的データ構造であることが可能である。スレッドが、移動と挿入のための呼出しを非同期的に実行することによって、そのスレッドの個々のコンポーネント内においてグローバルポインタｇ、ｈをランダムに移動させるだろう。これは、非常に高い弾力性を得るために、データ競合をインタリービング効果およびエイリアシング効果と組み合わせるという利点を有する。

図２７では、Ｖが１対のグローバル整変数Ｘ、Ｙである、よりはるかに単純な例を使用して、上記の着想を図解している。これは、任意の整数ｘおよび整数ｙの場合に「$7y^2-1$」がx^2に等しくないという基本数論からの公知の事実に基づいている。

９．混乱解除と予防変換

本発明者の混乱化変換の多く（特にセクション６．２の制御変換）は、実プログラム内にボーガスプログラム（ｂｏｇｕｓ　ｐｒｏｇｒａｍ）を埋め込むと言い表されることが可能である。言い換えれば、被混乱化アプリケーションは、実際には、１つに併合された２つのプログラムから成り、すなわち、有用なタスクを行う実プログラムと、無益な情報を計算するボーガスプログラムとが、１つに併合されている。このボーガスプログラムの唯一の目的は、無関係なコードの背後に実プログラムを隠蔽することによって、潜在的なリ

バースエンジニアを混乱させることである。
上記不明瞭述語は、ボーガス内部プログラムが容易に識別されて除去されることを防止するために、混乱化器が自由に使用できる主要な仕掛けである。例えば、図２８ａでは、混乱化器が、実プログラムの３つの文の中に、不明瞭述語によって保護されたボーガスコード（ｂｏｇｕｓ　ｃｏｄｅ）を埋め込む。混乱解除器のタスクは、被混乱化アプリケーションを調べて、内部ボーガスコードを自動的に識別して取り除くことである。これを行うために、混乱解除器が、最初に不明瞭構造体を識別してから、その構造体を評価しなければならない。このプロセスが図２８ｂから図２８ｄに示されている。
図２９は、半自動的混乱解除ツールの構造を示す。このツールは、リバースエンジニアリングのコミュニティでは公知である幾つかの手法を採り入れている。このセクションの残り部分では、こうした手法の幾つかを簡単に検討し、混乱解除をより困難にするために混乱化器を使用することが可能である様々な対抗策（いわゆる予防変換）を説明する。
９．１　予防変換
図２ｇに関連して上記で説明した予防変換は、制御変換またはデータ変換とは趣が全く異っている。制御変換またはデータ変換とは対照的に、予防変換の主要な目標は、人間の読み手に対してプログラムを覆い隠すことではない。むしろ、予防変換は、公知の自動混乱解除手法をより困難にするように（本来予防変換）、または、現在の混乱解除器またはデコンパイラにおける既知の問題を探り出すように（目標予防変換）設計される。
９．１．１　本来予防変換
本来予防変換は、一般的に、低い効力と高い弾力性とを有する。最も重要なことは、本来予防変換が他の変換の弾力性を増強する能力を有するだろうということである。一例として、セクション６．４で示唆されているように、ｆｏｒループを逆方向に実行するように再オーダリングし終わっているものと仮定する。ループがループ運搬データ従属性を持たないことを調べることが可能だったという理由だけから、この変換が適用されることが可能だった。当然のことながら、混乱解除器が同じ解析を行ってループを順方向実行に戻すことを阻止するものは何もない。これを防止するために、逆ループに対するボーガスデータ従属性を加えることが可能である。

```
{                                   {

for(i=1;i<=10;i++)   =T=>              int B[50];

     A[i]=i                            for(i=10;i<=1;i--)

}                                      A[i]=i;

                                       B[i]+=B[i*i/2]

                                    }
```

この本来予防変換がループ再オーダリング変換に加える弾力性は、ボーガス従属性の複雑性と従属性解析の技術的現状とに依存している［３６］。
９．１．２　目標予防変換
目標予防変換の一例として、ＨｏｓｅＭｏｃｈａプログラムを考察する（Ｍａｒｋ　Ｄ．ＬａＤｕｅ．ＨｏｓｅＭｏｃｈａ．ｈｔｔｐ：／／ｗｗｗ．ｘｙｎｙｘ．ｄｅｍｏｎ．ｎｌ／ｊａｖａ／ＨｏｓｅＭｏｃｈａ．ｊａｖａ，Ｊａｎｕａｒｙ　１９９７）。このプログラムは、Ｍｏｃｈａデコンパイラ（Ｈａｎｓ　Ｐｅｔｅｒ　Ｖａｎ　Ｖｌｉｅｔ．Ｍｏｃｈａ　－－－　Ｔｈｅ　Ｊａｖａ　ｄｅｃｏｍｐｉｌｅｒ．ｈｔｔｐ：／／ｗｅｂ．ｉｎｔｅｒ．ｎｌ．ｎｅｔ／ｕｓｅｒｓ／Ｈ．Ｐ．ｖａｎ．Ｖｌｉｅｔ／ｍｏｃｈａ．ｈｔｍｌ，Ｊａｎｕａｒｙ　１９９６）の弱点を調査するために特別に設計されている。ＨｏｓｅＭｏｃｈａプログラムは、ソースプログラム内の全ての方法の中のあらゆる復帰文の後に、特別な命令を挿入する。この変換は、アプリケーションの動作に対して全く影響を及ぼさないが、Ｍｏｃｈａをクラッシュさせるには十分である。

９．２　不明瞭構造体の識別と評価
混乱解除の最も重要な部分は、不明瞭構造体の識別と評価である。識別と評価が別個のアクティビティであることに留意されたい。不明瞭構造体は、ローカルである（単一の基本ブロック内に含まれる）か、または、グローバルである（単一の手続きに含まれる）か、または、手続き間である（プログラム全体にわたって分散している）ことが可能である。例えば、ｉｆ（ｘ＊ｘ＝＝（７^{F}＊ｙ＊ｙ－１））がローカル不明瞭述語であり、Ｒ＝Ｘ＊Ｘ；．．．；Ｓ＝７＊ｙ＊ｙ－１；．．．；ｉｆ（Ｒ＝＝Ｓ^{F}）．．．がグローバル不明瞭術語である。ＲとＳの計算が互いに異った手続きで行われた時には、構造体は手続き間不明瞭構造体であるだろう。当然のことながら、ローカル不明瞭述語の識別は、手続き間の不明瞭述語の識別よりも容易である。

９．３　パターン照合による識別
混乱解除器は、不明瞭述語を識別するために、既知の混乱化器によって使用されるストラレジの知識を使用することが可能である。混乱解除器の設計者は、（混乱化器をデコンパイルすることによって、または、単純に、混乱化器が生成する被混乱化コードを調査することによって）混乱化器を調査することが可能であり、一般的に使用される不明瞭述語を識別することが可能なパターン照合規則を構築することが可能である。この方法は、ｘ＊ｘ＝＝（７＊ｙ＊ｙ－１）またはｒａｎｄｏｍ（１^{F}，５）＜０のような単純なローカル述語に対して最も効果的に働くだろう。
パターン照合の試みを妨害するために、混乱化器は、常套的な不明瞭構造体の使用を避けなければならない。さらに、実アプリケーションで使用される構造体に構文上類似している不明瞭構造体を選択することも重要である。

９．４　プログラムスライシングによる識別
プログラマは、一般的に、プログラムの被混乱化バージョンが、リバースエンジニアにとってオリジナルのプログラムよりも理解が困難であることを発見するだろう。その主たる理由は、被混乱化プログラムでは、（ａ）生きている「実」コードに、死んだボーガスコードがちりばめられ、かつ、（ｂ）論理的に関係付けられたコード断片が分解されてプログラム全体にわたって分散されるということである。プログラムスライシングツールが、こうした混乱化に対抗するためにリバールエンジニアによって使用されることが可能である。こうしたツールは、スライスと呼ばれる管理可能なチャンク（ｃｈｕｎｃｋ）の形にプログラムを分解するために、リバースエンジニアを対話式に補助することが可能である。ポイントｐと変数ｖとに関するプログラムＰのスライスは、ｐにおいてｖの値に寄与することが可能だったＰの文の全てから成る。したがって、プログラムスライサは、混乱化器がこうした文をプログラム全体に分散させた場合にさえ、不明瞭変数ｖを計算するアルゴリズムの文を、被混乱化プログラムから抽出することが可能だろう。
スライシングを有効性がより劣る識別ツールにするために混乱化器で使用可能な幾つかのストラテジが存在する。ＡｄｄパラメタエイリアスＡパラメタエイリアスは、同じ記憶場所を参照する２つの仮パラメタ（または、仮パラメタとグローバル変数）である。厳密な手続き間スライシングは、プログラム中の潜在的エイリアスの個数に応じて増大し、一方、この潜在的エイリアスの個数は、仮パラメタの個数に応じて指数関数的に増大する。したがって、混乱化器が、エイリアシングされたダミーパラメタをプログラムに追加する場合には、その混乱化器が、（厳密なスライスが要求される場合には）スライサを著しく減速させるか、または、（高速スライシングが必要とされる場合には）スライサに非厳密なスライスを生じさせるように強制する。
Ｕｎｒａｖｅｌ（Ｊａｍｅｓ Ｒ．Ｌｙｌｅ，Ｄｏｌｏｒｒｅｓ Ｒ．Ｗａｌｌａｃｅ，Ｊａｍｅｓ Ｒ．Ｇｒａｈａｍ，Ｋｅｉｔｈ Ｂ．Ｇａｌｌａｇｈｅｒ，Ｊｏｓｅｐｈ Ｐ．Ｐｏｏｌｅ，ａｎｄ Ｄａｖｉｄ Ｗ．Ｂｉｎｋｌｅｙ．Ｕｎｒａｖｅｌ：Ａ ＣＡＳＥ ｔｏｏｌ ｔｏ ａｓｓｉｓｔ ｅｖａｌｕａｔｉｏｎ ｏｆ ｈｉｇｈ ｉｎｔｅｇｒｉｔｙ ｓｏｆｔｗａｒｅ．Ｖｏｌｕｍｅ １：Ｒｅｑｕｉｒｅｍｅｎｔｓ ａｎｄ ｄｅｓｉｇｎ．Ｔｅｃｈｎｉｃａｌ Ｒｅｐｏｒｔ ＮＩＳ－ＴＩＲ ５６９１，Ｕ．Ｓ．Ｄｅｐａｒｔｍｅｎｔ ｏｆ Ｃｏｍｍｅｒｃｅ，Ａｕｇｕｓｔ １９９５）のよ

うな一般に普及しているスライシングツールのような加算変数従属性（ａｄｄ　ｖａｒｉａｂｌｅ　ｄｅｐｅｎｄｅｎｃｙ）は、小さなスライスの計算には適切に機能するが、より大きいスライスの計算に対しては過大な時間を要する場合がある。例えば、４０００行のＣプログラムに対して使用した場合には、Ｕｎｒａｖｅｌがスライス計算を完了するのに３０分間を越える時間を必要としたケースがあった。こうした特徴を強制的に引き出すために、混乱化器が、ボーガス変数従属性を加えることによってスライスサイズを増大させようとしなければならない。次の例では、見掛け上ではｘの値に寄与するが実際には寄与しない２つの文を加えることによって、スライス計算のサイズｘが増大させられている。

```
main()          {          =T =>          main()  {
int x=1;                                   int x=1;
x = x * 3;                                      if (PF ) x++;
}                                          x = x + V0 ;
                                           x = x * 3;
                                           }
```

９．５　静的解析

混乱解除器は、述語全ての結果を分析するために、被混乱化プログラムを利用することが可能である。このようにして被混乱化アプリケーションの実行時の特徴を調査する任意の混乱解除方法、すなわち、静的解析が呼び出されるだろう。混乱解除器は、多数のテストランにおいて常に同じ真理値を戻す述語が不明瞭P^T（P^F）述語であることが判明するかも知れないので、こうした真理値を戻す述語の全てに対するリバースエンジニアの注意を促す。混乱解除器は、こうした述語を真（偽）でむやみに置き換えることは不可能であるが、これは、そうすることが危険過ぎる可能性があるからである。多くのアプリケーションは、例外的な状況下でしか発生しない条件を検査する「真の」述語を含むが、混乱解除器にとっては、こうしたアプリケーションは、不明瞭述語と同じに動作するように見えるだろう。例えば、ｐｉｆ（Ｌｅａｐ　Ｙｅａｒ）．．．を考察されたい。

静的解析は、さらに、評価のためにも使用されることが可能である。プログラムM内の潜在的な不明瞭述語（例えば、P^T）がすでに識別されている時に、その値（真）が推定され、上記不明瞭述語がこの推定値ですでに置き換えられている被混乱化プログラムのバージョンＭ′が作成される。その次に、同一の入力に対してＭとＭ′とが並行に実行され、ＭとＭ′とが同一の出力を生じさせることを確認するために比較が行われる。これらの出力が同一である場合には、図３０に示されているように、その述語が、真のアプリケーションではないボーガスの一部であったと結論付けられることが可能である。

選択された入力が上記プログラム中の全てのパスを適切にカバーすることが確保されなければならない。同様に、上記プログラムが、稀にしかとられないパス（ｉｆ（Ｌｅａｐ　Ｙｅａｒ）．．．）を含む場合には、これは困難だろう。さらに、多数の正しい入力／出力データを生成することは非常に困難であり、特にアプリケーションの内部構造が未知である時、または、（往々にしてＪａｖａプログラムに当てはまるように）複雑なグラフィカルユーザインタフェースを通して入力が行われる時には、これは非常に困難である。

静的解析による識別を防止するために、混乱化器が、P^T述語またはP^F述語を挿入する変換よりも、（図１１ｂに示されているような）$P^?$述語を挿入する変換を優先するように選択してもよい。

静的解析に対する別の採用可能な対抗策は、幾つかの述語が同時に解読されなければならないように、不明瞭述語を設計することである。これを行うための方法の１つは、不明瞭述語に副作用を持たせることである。下記の例では、混乱化器が、（一種の静的流れ解析によって）、文S_1、S_2が常に同じ回数だけ実行しなければならないということを決定する

。これらの文が、関数Q_1と関数Q_2とに対する呼出しである不明瞭述語を導入することによって混乱化される。関数Q_1と関数Q_2はグローバル変数を増減する。

```
{                  =T =>      {
S1;                            int k=0;
S2;                            bool Q1 (x)    {
}                              k+=2^i1 ; return (P^T 1)}
                               bool Q2 (x)    {
                               k-=2^i1 ; return (P^T 2)}
                              {
                               if (Q1 (j) T ) S1 ;
                                    . . .
                               if (Q2 (k) T ) S2 ;
                               }
```

混乱解除器が１つの（両方ではない）述語を真で置き換えようとする場合には、ｋがオーバフローするだろう。その結果として、混乱解除されたプログラムが、エラー状態で終了するだろう。

９．６　データフロー解析による評価

混乱解除は、様々なタイプのコード最適化に類似している。ｉｆ（Ｆａｌｓｅ）．．．を取り除くことは、死んだコード（ｄｅａｄ　ｃｏｄｅ）削除であり、ｉｆ文ブランチ（例えば、図２８におけるS_1とS_0[1]）から同一コードを移動することは、コードホイスティング（ｃｏｄｅ　ｈｏｉｓｔｉｎｇ）であり、これらは一般的なコード最適化手法である。

不明瞭構造体が識別され終わると、その構造体の評価を試みることが可能になる。単純な事例では、到達定義データフロー解析（ｒｅａｃｈｉｎｇ　ｄｅｆｉｎｉｔｉｏｎ　ｄａｔａ－ｆｌｏｗ　ａｎａｌｙｓｉｓ）を使用する定数伝搬で十分であることが可能である。ｘ＝５；．．．；ｙ＝７；．．．；ｉｆ（ｘ＊ｘ＝＝（７＊ｙ＊ｙ－１）．．．．

９．７　定理の証明による評価

データフロー解析が不明瞭述語を解読するのに十分なだけ強力ではない場合には、混乱解除器が定理の証明を使用することを試みることが可能である。これが可能であるか不可能であるかは、（確認が困難な）技術的現状の定理証明プログラムの能力と、証明されることが必要な定理の複雑性とに依存している。当然のことながら、帰納によって証明可能な定理（例えば、$x^2(x+1)^2=0 \pmod 4$）は、十分に現行の定理証明プログラムの到達範囲内である。

事柄をさらに困難にするために、証明が困難であることが知られている定理、または、それに関する既知の証明が存在しない定理を使用することが可能である。下記の例では、混乱解除器が、S_2が生きたコード（ｌｉｖｅ　ｃｏｄｅ）であることを調査するために、ボーガスループ（ｂｏｇｕｓ　ｌｏｏｐ）が常に終了するということを証明しなければならないだろう。

```
{              =T =>      {
S1;                       S1;
S2;                       n = random(1,2^32);
}                         do
                               n = ((n%2)!=0)?3*n+1:n/2
                          while (n>1);
                          S2;
                           }
```

これは、Ｃｏｌｌａｔｚ問題として知られている。上記ループが常に終了するだろうということが推測される。この推測の既知の根拠は存在しないが、７＊１０^{11}までの全ての数に関してそのコードが終了することが知られている。したがって、この混乱化は安全であり（オリジナルの混乱化されたコードが全く同じに動作する）、混乱解除を行うことは困難である。

９．８　混乱解除および部分評価

混乱解除は、さらに、部分評価にも類似している。部分評価器は、プログラムを、２つの部分、すなわち、部分評価器によって事前計算されることが可能である静的部分と、実行時に実行される動的部分とに分割する。動的部分は、混乱化されていないオリジナルのプログラムに相当するだろう。静的部分は、ボーガス内部プログラムに相当し、このボーガス内部プログラムは、識別された場合には、混乱解除時点で評価され除去されることが可能である。

他の静的内部手続き解析手法の全てと同様に、部分評価はエイリアシングの影響を受けやすい。したがって、スライシングに関連して言及した予防変換と同じ予防変換が、部分評価にも適用される。

１０．混乱化アルゴリズム

次に、セクション３の混乱化器アーキテクチャと、セクション５の混乱化品質の定義と、セクション６からセクション９の様々な混乱化変換の説明とに基づいて、本発明の実施様態の１つによるさらに詳細なアルゴリズムを説明する。

混乱化ツールの最上位レベルのループは、次の一般構造を有することが可能である。

```
WHILE NOT Done(A) DO
     S := SelectCode(A);
     T := SelectTransform(S);
     A := Apply(T ,S);
END;
```

ＳｅｌｅｃｔＣｏｄｅは、混乱化されるべきその次のソースコードオブジェクトを戻す。ＳｅｌｅｃｔＴｒａｎｓｆｏｒｍは、この特定のソースコードオブジェクトを混乱化するために使用されなければならない変換を戻す。Ａｐｐｌｙが、変換をソースコードオブジェクトに適用し、それに応じてアプリケーションを更新する。Ｄｏｎｅは、所要レベルの混乱化に達し終わった時を決定する。これらの関数の複雑性は、混乱化ツールのソフィスティケーションに依存するだろう。単純な位取りの結果として、ＳｅｌｅｃｔＣｏｄｅとＳｅｌｅｃｔＴｒａｎｓｆｏｒｍとが単純にランダムなソースコードオブジェクト／変換を戻し、Ｄｏｎｅが、特定の限界をアプリケーションのサイズが越える時にループを終了

させることが可能である。通常では、こうした動作は不十分である。
アルゴリズム１は、さらにはるかに洗練された選択動作および終了動作を含むコード混乱化ツールの記述を与える。実施様態の１つでは、このアルゴリズムが幾つかのデータ構造を使用し、こうしたデータ構造がアルゴリズム５、６、７によって構築される。
各ソースコードオブジェクトＳに関するP_Sである時に、P_S（Ｓ）は、プログラマがＳで使用した言語構造体セットである。P_S（Ｓ）が、Ｓに関する適切な混乱化変換を発見するために使用される。
各ソースコードオブジェクトＳに関するＡである時に、Ａ（Ｓ）＝｛T_i－－＞　V_1；．．．；T_n　－－＞V_n｝は、変換T_iから値V_iへのマッピングであり、T_iをＳに適用することがどのように適切であるかを記述する。この着想は、Ｓにとって「不自然である」新たなコードを特定の変換が導入するので、こうした変換が特定のソースコードオブジェクトＳに関して不適切である可能性があるということである。この新たなコードは、Ｓの中では場違いに見え、したがって、リバースエンジニアには発見しやすいだろう。適切性値（ａｐｐｒｏｐｒｉａｔｅｎｅｓｓ　ｖａｌｕｅ）V_iが大きければ大きいほど、変換T_iによって導入されるコードがより良好に適合するだろう。
各ソースコードオブジェクトＳに関するＩである時に、Ｉ（Ｓ）はＳの混乱化プライオリティである。Ｉ（Ｓ）が、Ｓの内容を混乱化することがどれだけ重要であるかを記述する。Ｓが重要なトレードシークレットを含む場合には、Ｉ（Ｓ）がＨＩＧＨであるだろうし、一方、それが主として「ブレッドアンドバター」コードを含む場合には、Ｉ（Ｓ）がＬＯＷだろう。
各ルーチンＭに関するＲである時に、Ｒ（Ｍ）はＭの実行時間ランクである。他のいずれのルーチンよりも多くの時間がＭを実行するために費やされる場合には、Ｒ（Ｍ）＝１である。
アルゴリズム１に対する一次入力は、アプリケーションＡと混乱化変換のセット｛T_1；T_2；．．．｝である。このアルゴリズムは、さらに、各々の変換に関する情報も要求し、特に、（セクション５での同名の関数と同様であるが、数値を戻す）３つの品質関数T_{res}（Ｓ）、T_{pot}（Ｓ）、T_{cost}（Ｓ）と、関数P_tを要求する。
ソースコードオブジェクトＳに適用される時に、T_{res}（Ｓ）が、変換Ｔの弾力性（すなわち、Ｔが自動混乱解除器からの攻撃にどれだけ適切に耐えるか）の測度を戻す。
ソースコードオブジェクトＳに適用される時に、T_{pot}（Ｓ）が、変換Ｔの効力（すなわち、Ｔによって混乱化された後に、人間がどれほど難しいＳを理解するのか）の測度を戻す。
T_{cost}（Ｓ）が、ＴによってＳに加えられる実行時間および空間ペナルティーの測度を戻す。
P_tが、Ｔがアプリケーションに加えることになる言語構造体のセットに各々の変換Ｔをマップする。
アルゴリズム１のポイント１からポイント３が、混乱化されるべきアプリケーションをロードし、適切な内部データ構造を構築する。ポイント４が、P_S（Ｓ）、Ａ（Ｓ）、Ｉ（Ｓ）、Ｒ（Ｍ）を構築する。所要の混乱化レベルに達し終わるまで、または、最大実行時間ペナルティが越えられるまで、ポイント５が混乱化変換を適用する。最後に、ポイント６が新たなアプリケーションＡ′を書き換える。
アルゴリズム１（コード混乱化）
入力：
ａ）ソースコードまたはオブジェクトコードファイルで構成されているアプリケーションＡ　Ｃ１；Ｃ２；．．．。
ｂ）言語によって定義される標準ライブラリＬ１；Ｌ２；．．．。
ｃ）混乱化変換セット｛Ｔ１；Ｔ２；．．．｝。
ｄ）変換Ｔの各々に関して、Ｔがアプリケーションに加えることになる言語構造体セットを与えるマッピングＰｔ。
ｅ）ソースコードオブジェクトＳに対する変換Ｔの品質を表現する３つの関数T_{res}（Ｓ

）、T_{pot}（Ｓ）、T_{cost}（Ｓ）。
ｆ）Ａに対する入力データセットＩ＝｛Ｉ１；Ｉ２；．．．｝。
ｇ）２つの数値ＡｃｃｅｐｔＣｏｓｔ＞０およびＲｅｑＯｂｆ＞０。ＡｃｃｅｐｔＣｏｓｔが、ユーザが受け入れることになる最大の追加実行時間／空間ペナルティの測度である。ＲｅｑＯｂｆが、ユーザによって要求される混乱化の量の測度である。
出力：　ソースコードまたはオブジェクトコードファイルで構成されている被混乱化アプリケーションＡ′。
１．混乱化されるべきアプリケーションC_1；C_2；．．．をロードする。混乱化器は、
（ａ）ソースコードファイルをロードすることが可能であり、この場合には混乱化器が、字句解析と構文解析と意味解析とを行う完全なコンパイラフロントエンドを含まなければならないだろうし（純粋に構文変換だけに自己限定する非力な混乱化器は、意味解析なしに処理することが可能である）、または、
（ｂ）オブジェクトコードファイルをロードすることが可能であり、オブジェクトコードがソースコード内に情報の大半または全てを保持する場合には（Ｊａｖａクラスファイルの場合のように）この方法が好ましい。
２．アプリケーションによって直接的または間接的に参照されるライブラリコードファイルＬ１；Ｌ２；．．．をロードする。
３．アプリケーションの内部表現を構築する。内部表現の選択は、混乱化器が使用するソース言語の構造と変換の複雑性とに依存する。典型的なデータ構造セットは、次のものを含んでよい。
（ａ）Ａ内の各ルーチンに関する制御流れグラフ。
（ｂ）Ａ内のルーチンに関する呼出しグラフ。
（ｃ）Ａ内のクラスに関する継承グラフ。
４．マッピングＲ（Ｍ）およびP_s（Ｓ）（アルゴリズム５を使用）、Ｉ（Ｓ）（アルゴリズム６を使用）、及びＡ（Ｓ）（アルゴリズム７を使用）を構築する。
５．アプリケーションに混乱化変換を適用する。各ステップにおいて、混乱化されるべきソースコードオブジェクトＳと、Ｓに対して適用すべき適切な変換Ｔとが選択される。所要の混乱化レベルに達した時に、または、許容可能な実行時間コストを超過した時に、このプロセスが終了する。
ＲＥＰＥＡＴ
Ｓ　：＝　ＳｅｌｅｃｔＣｏｄｅ（Ｉ）；
Ｔ　：＝　ＳｅｌｅｃｔＴｒａｎｓｆｏｒｍ（Ｓ，Ａ）；
ＴをＳに適用し、ポイント３からの適切なデータ構造を更新する；
ＵＮＴＩＬ　Ｄｏｎｅ（ＲｅｑＯｂｆ，ＡｃｃｅｐｔＣｏｓｔ，Ｓ，Ｔ，Ｉ）．
６．被混乱化ソースコードオブジェクトを新たな被混乱化アプリケーションＡ′に再構成する。
アルゴリズム２（ＳｅｌｅｃｔＣｏｄｅ）
入力：　アルゴリズム６によって計算される通りの混乱化プライオリティマッピング。
出力：　ソースコードオブジェクトＳ。
Ｉが各々のソースコードオブジェクトＳをＩ（Ｓ）にマップし、このＩ（Ｓ）は、Ｓを混乱化することがどれだけ重要であるかの測度である。混乱化すべき次のソースコードオブジェクトを選択するために、Ｉが優先待ち行列として処理されることが可能である。言い換えれば、Ｉ（Ｓ）が最大化されるようにＳが選択される。
アルゴリズム３（ＳｅｌｅｃｔＴｒａｎｓｆｏｒｍ）
入力：
ａ）ソースコードオブジェクトＳ。
ｂ）アルゴリズム７によって計算される通りの適切性マッピングＡ。
出力：　変換Ｔ
ある特定のソースコードオブジェクトＳに適用するための最も適切な変換を選択するために、任意の個数の発見法（ｈｅｕｒｉｓｔｉｃｓ）が使用可能である。しかし、考慮すべ

き２つの重要な問題がある。第１に、選択された変換が、Ｓ内のコードの残り部分と自然に混じり合わなければならない。これは、Ａ（Ｓ）において高い適切性値を有する変換を優先することによって取扱可能である。第２に、高い「出費に見合うだけの価値（ｂａｎｇ－ｆｏｒ－ｔｈｅ－ｂｕｃｋ）」を得る（すなわち、低い実行時間ペナルティで高レベルの混乱化を得る）変換が優先されることが望ましい。これは、効力と弾力性を最大化すると同時にコストを最小化する変換を選択することによって実現される。これらの発見法が次のコードによって取り込まれ、このコード内のｗ１、ｗ２、ｗ３がインプリメンテーション定義定数である。

Ｔ　－－＞　ＶがＡ（Ｓ）内であるように変換Ｔを戻し、かつ、（ｗ１＊T_{pot}ｔ（Ｓ）＋ｗ２＊T_{res}（Ｓ）＋Ｗ３＊Ｖ）／T_{cost}（Ｓ）が最大化される。

アルゴリズム４（Ｄｏｎｅ）

入力：

ａ）ＲｅｑＯｂｆ、混乱化の残留レベル。

ｂ）ＡｃｃｅｐｔＣｏｓｔ、残留する許容可能な実行時間ペナルティ。

ｃ）ソースコードオブジェクトＳ。

ｄ）変換Ｔ。

ｅ）混乱化プライオリティマッピングＩ。

出力：

ａ）更新されたＲｅｑＯｂｆ。

ｂ）更新されたＡｃｃｅｐｔＣｏｓｔ。

ｃ）更新された混乱化プライオリティマッピングＩ。

ｄ）終了条件に達している場合には真であるブール返却値。

Ｄｏｎｅ関数が２つの目的を満たす。この関数は、ソースコードオブジェクトＳが混乱化され終わっており、かつ、縮小された優先順位値を受け取らなければならないという事実を反映させるために、優先待ち行列Ｉを更新する。この縮小は、変換の弾力性と効力との組合せに基づいている。Ｄｏｎｅは、さらに、ＲｅｑＯｂｆとＡｃｃｅｐｔＣｏｓｔとを更新し、終了条件に達しているかどうかを調べる。ｗ１、ｗ２、ｗ３、ｗ４は、インプリメンテーション定義定数である。

$I(S) := I(S) - (w_2T_{pos}(S) + w_2T_{res}(S))$;

$ReqObf := ReqObf - (w_2T_{pos}(S) + w_2T_{res}(S))$;

$AcceptCost := AcceptCost - T_{cost}(S)$;

RETURN AcceptCost <= 0 OR ReqObf <= 0.

アルゴリズム５（プラグマティック情報）

入力：

ａ）アプリケーションＡ。

ｂ）Ａに対する入力データセットＩ＝｛Ｉ１；Ｉ２；．．．｝。

出力：

ａ）Ａ内の各ルーチンＭに関してＭの実行時間ランクを与えるマッピングＲ（Ｍ）。

ｂ）Ａ内の各ソースコードオブジェクトＳに関してＳで使用される言語構造体のセットを与える、マッピングＰｓ（Ｓ）。

プラグマティック情報を計算する。この情報が、各々の個別のソースコードオブジェクトに関する情報の適正タイプを選択するために使用されることになる。

１．動的プラグマティック情報を計算する（すなわち、ユーザによって提供される入力データセットＩに基づくプロファイラ下でアプリケーションを実行する）。アプリケーションがその時間の大部分をどこで費やすのかを示す、各ルーチン／基本ブロックに関するＲ（Ｍ）（Ｍの実行時間ランク）を計算する。

２．静的プラグマティック情報P_S（Ｓ）を計算する。P_S（Ｓ）が、プログラマがＳで使用した言語構造体の種類に関する統計を提供する。

ＦＯＲ　Ｓ　：＝　Ａ　ＤＯにおける各ソースコードオブジェクト

Ｏ　：＝　Ｓが使用する演算子のセット；

Ｃ　：＝　Ｓが使用する高レベル言語構造体（ＷＨＩＬＥ文、例外、スレッド等）のセット；

Ｌ　：＝　Ｓが参照するライブラリクラス／ルーチンのセット；

Ｐｓ（Ｓ）：＝　Ｏ　Ｕ　Ｃ　Ｕ　Ｌ；

ＥＮＤ　ＦＯＲ.

アルゴリズム６（混乱化プライオリティ）
入力：
ａ）アプリケーションＡ。
ｂ）Ｒ（Ｍ）、Ｍのランク。
出力：
Ａの各ソースコードオブジェクトＳに関してＳの混乱化プライオリティを与えるマッピングＩ（Ｓ）。
Ｉ（Ｓ）が、ユーザによって明示的に提供されることが可能であるか、または、アルゴリズム５で収集された静的データに基づいた発見法を使用して計算されることが可能である。使用可能な発見法は次の通りであってよい。
１．Ａの任意のルーチンＭに関して、Ｍのランク、すなわち、Ｒ（Ｍ）に対してＩ（Ｍ）を反比例させる。すなわち、この着想は、「ルーチンを実行するのに多くの時間が費やされる場合には、恐らくは、Ｍが、厳重に混乱化されなければならない重要な手続きだろう」ということである。
２．テーブル１のソフトウェア複雑性測度の１つによって定義される通りに、Ｉ（Ｓ）をＳの複雑性とする。この場合も同様に、（間違っている可能性もある）直感的洞察は、複雑なコードの方が単純なコードの場合よりも重要なトレードシークレットを含む可能性が高いということである。
アルゴリズム７（混乱化の適切性）
入力：
ａ）アプリケーションＡ。
ｂ）変換Ｔの各々に関して、Ｔがアプリケーションに加えるであろう言語構造体のセットを与える、マッピングＰｔ。
ｃ）Ａの各ソースコードオブジェクトＳに関して、Ｓで使用される言語構造体のセットを与えるマッピングＰｓ（Ｓ）。
出力：
Ａの各ソースコードオブジェクトＳと各変換Ｔとに関して、Ｓに対するＴの適切性を与えるマッピングＡ（Ｓ）。
各ソースコードオブジェクトＳに関して適切性セットＡ（Ｓ）を計算する。マッピングは、基本的に、アルゴリズム５で計算された静的プラグマティック情報に基づいている。

```
ＦＯＲ　Ｓ　：＝　Ａ　ＤＯにおける各ソースコードオブジェクト
    ＦＯＲ　Ｔ　：＝　各変換ＤＯ
        Ｖ　：＝　Ｐｔ（Ｔ）とＰｓ（Ｓ）との間の類似
                性の度合い；
        Ａ（Ｓ）：＝　Ａ（Ｓ）　Ｕ　｛Ｔ　－－＞　Ｖ
                ｝；
    ＥＮＤ　ＦＯＲ
ＥＮＤ　ＦＯＲ
```

１１．概要と考察

被混乱化プログラムがオリジナルのプログラムとは異った形で動作することが、多くの状況において許容可能であるだろうということを、本発明者は認識している。特に、本発明の混乱化変換の殆どが、そのオリジナルのプログラムに比べて目標プログラムの動作速度を遅くするかまたはプログラムサイズを大きくする。本発明では、特殊な場合に、目標プログラムが、オリジナルのプログラムとは異った副作用を有することさえ可能とされ、または、オリジナルのプログラムがエラー条件で終了する時に目標プログラムが終了しないことさえ可能とされる。本発明の混乱化変換の唯一の必要条件は、これら２つのプログラムの観測可能な動作（ユーザによって経験されるような動作）が同一でなければならないということだけである。

オリジナルのプログラムと被混乱化プログラムとの間のこうした弱い同一性を可能にすることは、新規性のある非常に刺激的な着想である。様々な変換が提供され上述されているが、他の様々な変換が当業者には明らかだろうし、本発明によるソフトウェアセキュリティ増強のための混乱化を実現するために使用されることが可能である。

さらに、未だ知られていない変換を発見するために、様々な将来の調査研究が行われる可能性が大きい。特に、次の領域の調査研究が行われることが期待されている。

１．新たな混乱化変換が発見されるべきである。

２．様々な変換の間での相互関係とオーダリングとが研究されるべきである。これは、最適化変換シーケンスのオーダリングが常に困難な問題である、コード最適化における研究と同様である。

３．効力とコストとの間の関係が研究されるべきである。個々の種類のコードに関して、どの変換が最良の「出費に見合うだけの価値」（すなわち、最低の実行オーバヘッドで最高の効力）を与えるかを知ることが求められている。

上記の変換の全てを概観するためには、図３１を参照されたい。上記の不明瞭構造体を概観するためには、図３２を参照されたい。しかし、本発明は、上記の典型的な変換と不明瞭構造体とに限定されてはならない。

１１．１　混乱化能力

暗号化とプログラム混乱化は互いに極めて類似している。これらの両方は、秘密を探り出そうとする人間たちの目から情報を隠蔽しようとするだけでなく、限られた時間でこの隠蔽を行うことも意図している。暗号化されたドキュメントは限られた貯蔵寿命を有する。暗号化プログラム自体が攻撃に耐えている限りにおいてだけ、かつ、ハードウェアの処理速度の進歩が、選択されたキー長に関するメッセージが頻繁に解読されることを可能にしない限りにおいてだけ、暗号化されたドキュメントが安全であるにすぎない。被混乱化アプリケーションにも同じことが当てはまる。十分に強力な混乱解除器が未だ構築されていない限りにおいてだけ、被混乱化アプリケーションが安全であるにすぎない。

混乱解除器が混乱化器が追いつくのに要する時間よりもリリースの間の時間が短い限りは

、アプリケーションを進化させる上では、このことは問題ではないだろう。混乱解除器が混乱化器に追いついても、アプリケーションが自動的に混乱解除されることが可能となる時点までには、そのアプリケーションが既に時代遅れになっており、したがって競合相手の関心の対象ではなくなっているだろう。
しかし、アプリケーションが、何回かのリリースにわたって存続すると考えられるトレードシークレットを含む場合には、こうしたトレードシークレットが混乱化以外の手段によって保護されなければならない。部分サーバ側実行（図２（ｂ））がその自明の選択であるが、アプリケーションの実行速度が低速となるかまたは（ネットワーク接続がダウンした時には）実行が不可能となるという欠点がある。

１１．２　混乱化の他の使用

上記の通りの混乱化用途以外に他の潜在的な混乱化用途が存在するかも知れないということを指摘することは興味深い。１つの可能性は、ソフトウェアの海賊版製造販売業者を追跡するために混乱化を使用することである。例えば、ベンダが、自分のアプリケーションの新たな被混乱化バージョンを新たな個々の顧客のために作成し（ＳｅｌｅｃｔＴｒａｎｓｆｏｒｍアルゴリズム（アルゴリズム３）の中にランダム性要素を導入することによって、同じアプリケーションの各々に異った被混乱化バージョンが生成されることが可能である。乱数発生器に対する異ったシードが別々のバージョンを生じさせる）、各バージョンを販売した顧客の記録を保持する。そのアプリケーションがネットを通じて販売され配給されている場合にだけ、これはおそらく妥当であるにすぎないだろう。ベンダが自分のアプリケーションの海賊版が販売されていることを発見した場合には、このベンダに必要なことは、その海賊版バージョンのコピーを手に入れて、それをデータベースと比較し、そのオリジナルのアプリケーションを誰が購入したかを発見することだけである。実際は、販売された全ての被混乱化バージョンのコピーを保存することは不要である。販売された乱数シードを保存しておくだけで十分である。
ソフトウェアの海賊版製造販売業者が混乱化を（不正）使用する可能性もある。本明細書で概説されたＪａｖａ混乱化器がバイトコードのレベルで動作するので、合法的に購入されたＪａｖａアプリケーションに対して海賊版製造販売業者が混乱化を施すことを阻止するものは何もない。その後で、この被混乱化バージョンが再販売されることも可能である。海賊版製造販売業者が訴訟に直面した時には、彼らは、実際には自分が最初に購入したアプリケーションを再販売しているのではなくて（結局のところ、そのコードが全く異っている！）、合法的に再設計したバージョンを販売しているのであると主張することも可能である。

結　論

結論として、本発明は、ソフトウェアのリバースエンジニアリングを防止するかまたは少なくとも妨害するための、コンピュータ上で実現される方法と装置とを提供する。これは、実行時間またはプログラムサイズを犠牲として実現されることが可能であり、その結果として変換されたプログラムが詳細なレベルでは異った形で動作するけれども、本発明の方法が、適切な状況では高い有用性をもたらすものと考えられる。実施様態の１つでは、変換されたプログラムが、非変換のプログラムと見掛け上では同じ動作をする。したがって、本発明は、オリジナルのプログラムと被混乱化プログラムとの間のこうした弱い同一性を可能にする。
本発明の開示が主としてソフトウェアのリバースエンジニアリングの阻止という文脈において行われてきたが、ソフトウェアオブジェクト（アプリケーションを含む）のウォーターマーキング（ｗａｔｅｒｍａｒｋｉｎｇ）ような他のアプリケーションが想定されている。これは、あらゆる単一の混乱化手続きの潜在的に特有の性質を利用する。ベンダは、アプリケーション販売先の個々の顧客に対して、個々に異った被混乱化バージョンを作成するだろう。海賊版コピーが発見された場合には、そのベンダは、その海賊版バージョンをオリジナルの混乱化情報データベースと比較するだけで、オリジナルのアプリケーションを追跡することが可能である。
本明細書で説明されている特定の混乱化変換は、網羅的なものではない。さらに別のタイ

プの混乱化が、本発明の新規の混乱化ツールアーキテクチャにおいて提供され使用されてよい。

上記の説明では、公知の等効物を有する要素または整数が言及されてきたが、こうした等効物が、上記で個々に説明されたように、本発明に含まれている。

本発明が特定の実施様態を参照して単なる具体例の形で説明されてきたが、本発明の範囲から逸脱することなしに、改変と改善とが行われることが可能であることが理解されなければならない。

【図１】

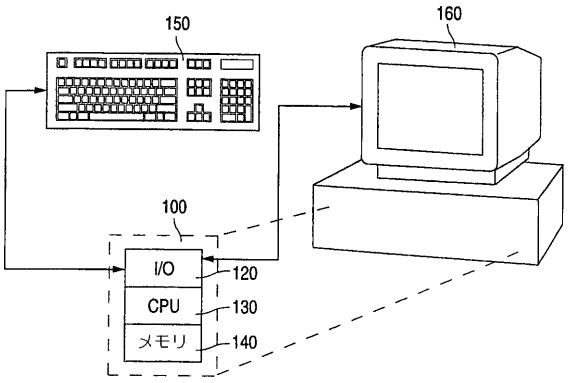
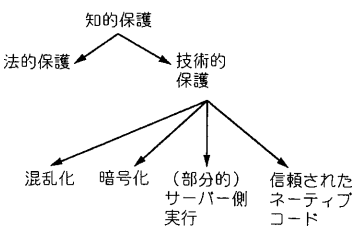


FIG. 1

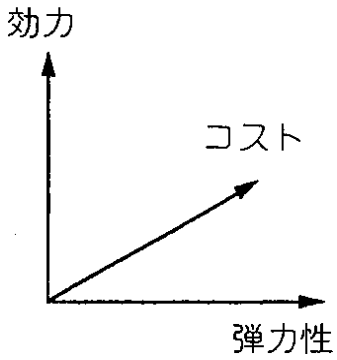
【図２ａ】

FIG. 2a



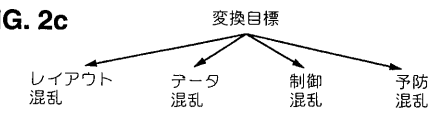
【図２ｂ】

FIG. 2b

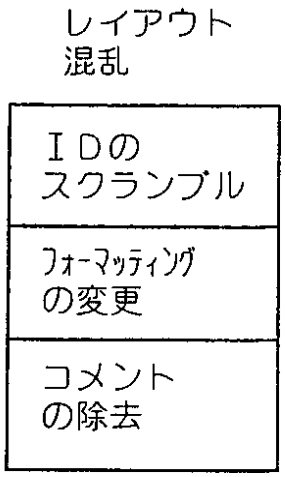


【図２ｃ】

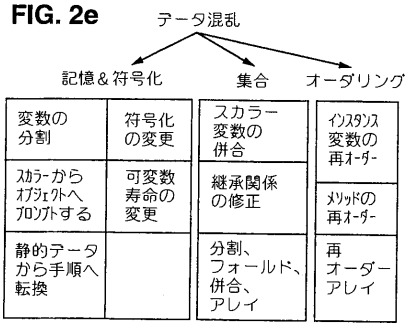
FIG. 2c



【図２ｄ】
FIG. 2d



【図２ｅ】
FIG. 2e



【図２ｆ】

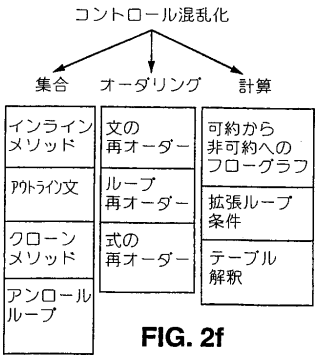


FIG. 2f

【図２ｇ】

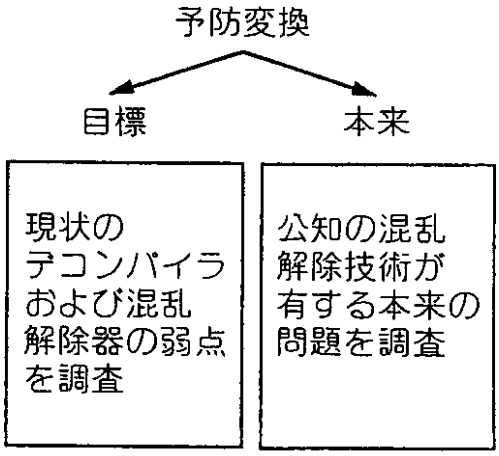


FIG. 2g

【図３ａ】

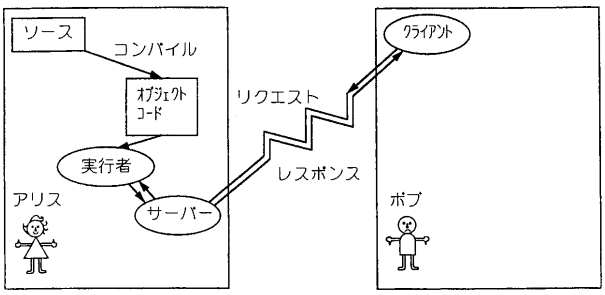


FIG. 3a

【図３ｂ】

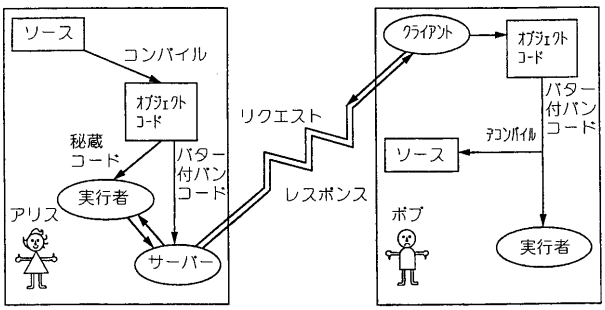


FIG. 3b

【図４ａ】

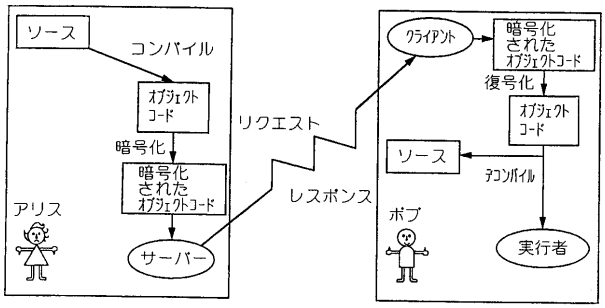


FIG. 4a

【図５】

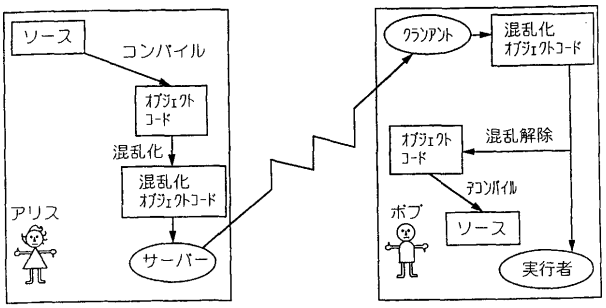


FIG. 5

【図４ｂ】

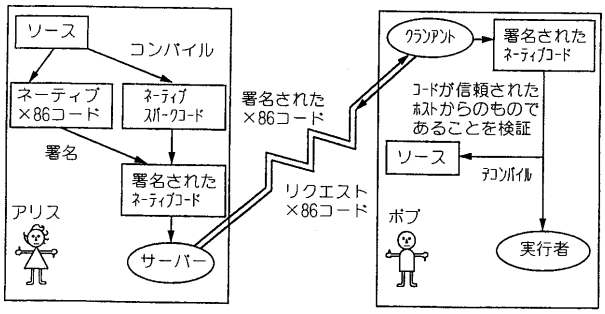


FIG. 4b

【図６】

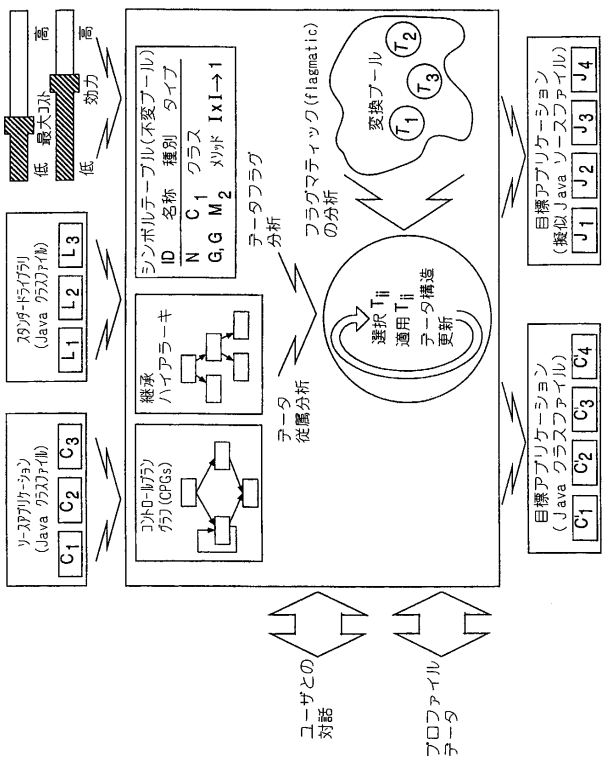


FIG. 6

【図７】

メトリック	メトリック名称	引用
μ_1	プログラム長	Halstead
	E(P)は、P内のオペランドおよび演算子の数と共に増大する	
μ_2	ｻｲｸﾛﾏﾃｨｯｸ(cyclomatic)の複雑性	McCabe
	E(F)は、F内の述語の数と共に増大する	
μ_3	入れ子の複雑性	Harrison
	E(F)は、F内条件の入れ子レベルと共に増大する	
μ_4	データフローの複雑性	Oviedo
	E(F)は、F内の内部基本ブロック可変レファレンスの数と共に増大する	
μ_5	ファンーイン／アウトの複雑性	Henry
	E(F)は、Fへの形式パラメータの数と共に、そして、Fにより読まれまたは更新されるグローバルデータ構造の数と共に、増大する	
μ_6	データ構造の複雑性	Munson
	E(P)は、P内にて宣言された静的データ構造の複雑性と共に、増大する。スカラー変数の複雑性は不変である。アレイの複雑性は、ディメンション数と共に、そして、エレメントタイプの複雑性と共に、増大する。レコードの複雑性は、そのフィールドの数および複雑性と共に、増大する。	
μ_7	OOメトリック	Chidamber
	E(C)は、C_1内のメソッド数(μ_7^a)と、継承木構造内のCの深さ(ルートからの距離)(μ_7^b)と、C_1のダイレクトサブクラスの数(μ_7^c)と、Cが結合される他のクラスの数(μ_7^d)と、C_1のオブジェクトへ送られたメッセージに応答して実行可能なメソッド数(μ_7^e)と、インスタンス変数の同一セットをCのメソッドが参照しない度合(μ_7^f)と共に、増大する。注：μ_7^fは結合度、すなわち、モジュールのエレメントがどれ位強く関連しているか、を測る。	

*2つのクラスは、もしその一方が他方のメソッドまたはインスタンス変数を使用するならば、結合される。

FIG. 7

【図８ａ】

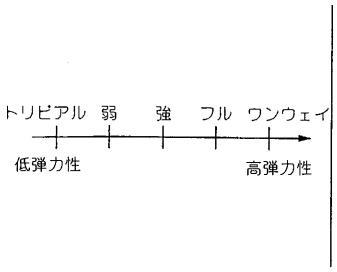


FIG. 8a

【図８ｂ】

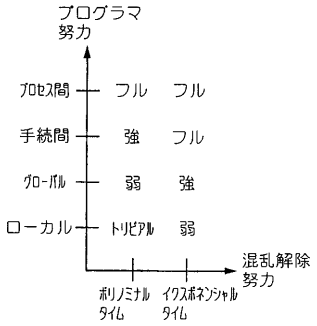


FIG. 8b

【図９】

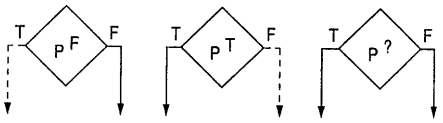


FIG. 9

【図１０ａ】

```
{     int v, a=5; b=6;
      v=11 = a + b;
      if (b > 6) T ...
      if (random (1,5) < 0) F...
}
```

FIG. 10a

【図１０ｂ】

```
{     int v, a=5; b=6;
      if (...) ...
           ⋮      (b is unchanged)
      if (b < 7) T  a++1
      v=11 = (a > 5) 7v=b=b; v=b
}
```

FIG. 10b

【図１１】

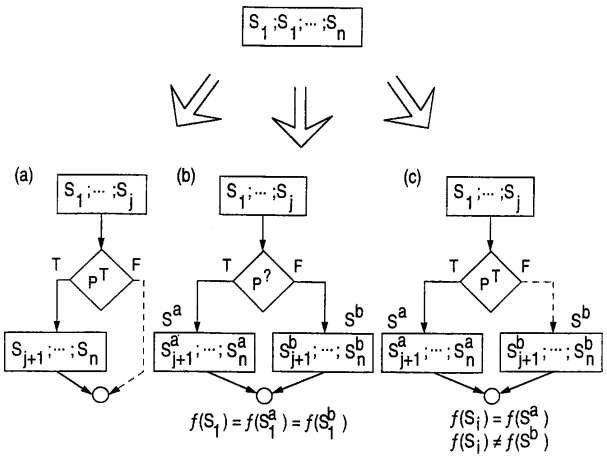


FIG. 11

【図１２】

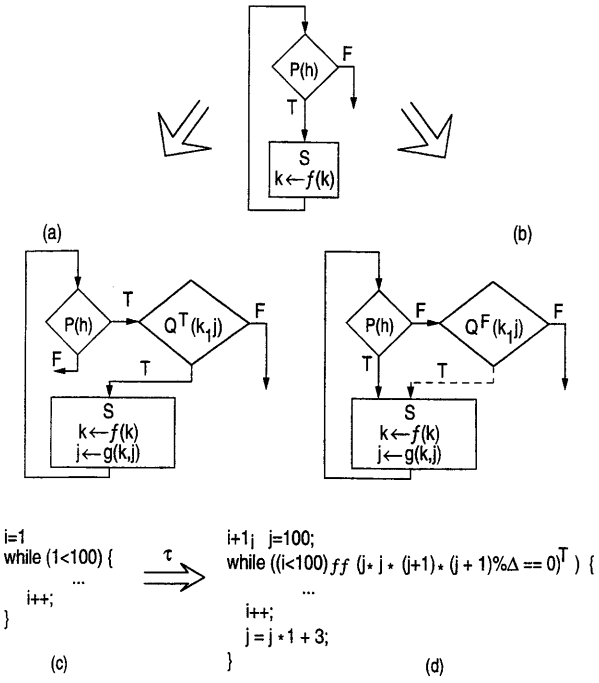


FIG. 12

【図１３】

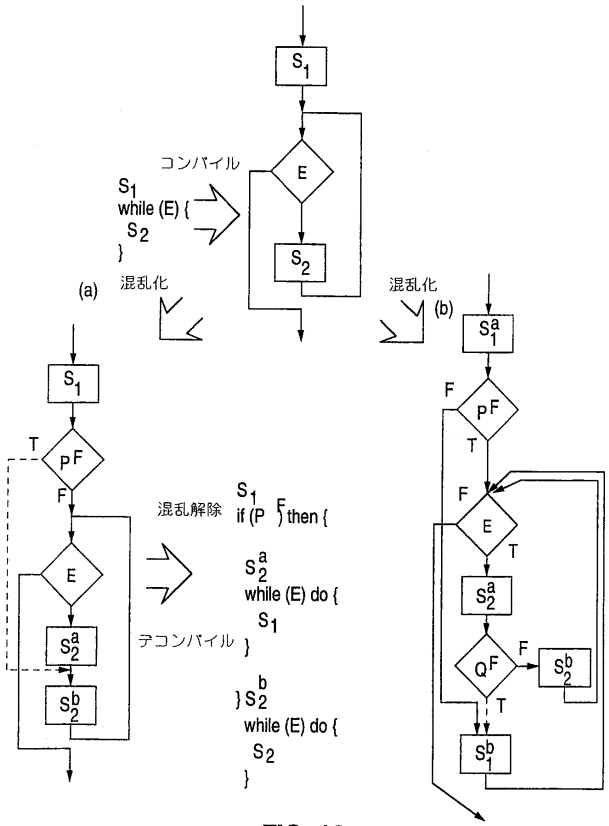


FIG. 13

【図１４】

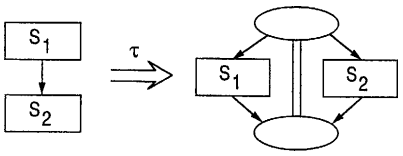


FIG. 14

【図１５】

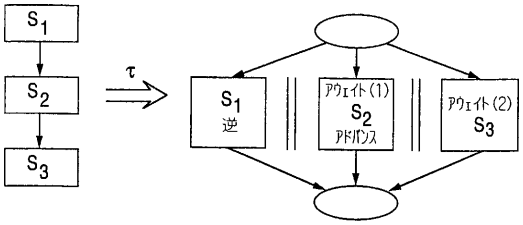


FIG. 15

【図１６】

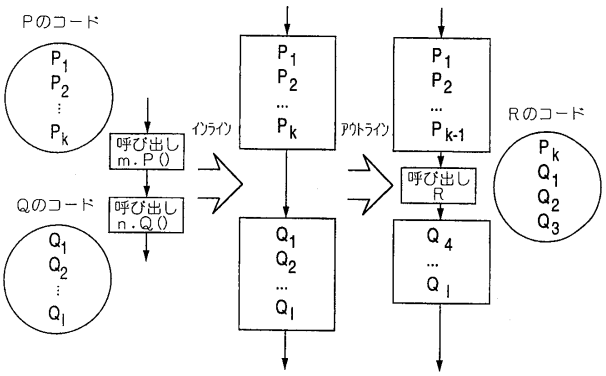


FIG. 16

【図１７】

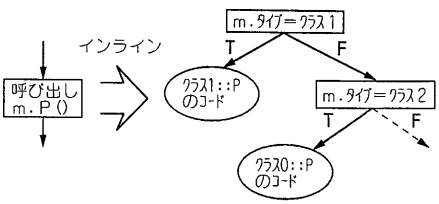


FIG. 17

【図１８】

```
class C {
  method M1 (T1 a) {
    S M1_1 ;... S M1_k ;
  }
  method M2 (T1 b; T2 c) {
    S k1_1 ;... S k2_m ;
  }
}

{ C x=new C;
  x.M1(a); x.M2(b, c); }
```

τ ⟹

```
class C' {
  method M (Ti a; T2 c; int V) {
    if (V==p) {S M1_1 ;...S M1_k ;}
    else      {S M1_1 ;...S k2_m ;}
  }
}

{ C' x=new C';
  x.M(a, c, V=p);
  x.M(b, c, V=g); }
```

FIG. 18

【図１９】

```
class C {
  method m (int x)
    {S_1 ... S_k}
}

{ C x = new C;
  x.m(8); ... x.m(7);
}
```

τ ⟹

```
class C1 {
  method m (int x)
    {S a_1 ;...S a_n }
  method m1 (int x)
    {S c_1 ;...S c_n }
}
class 02 inherits C1 {
  method M (int x)
    {S b_1 ... S b_k }
}

{ C1 x ;
  if (P7) x=new C1 else x=new C2;
  x.m(5); ...; x.m1(7);
}
```

FIG. 19

【図２０ａ】

FIG. 20a

```
for (i=1,i<=n,i++)
  for (j=1, j<=n,j++)
    a[1,j]=b[j,i]
```

τ ⟹

```
for(I=1, I<=n, I+=64)
  for (J=1,J<=n,J+=64)
    for (i=I,i<=min(I+63,n),i++)
      for (j_J,j<=min(J+65,n),j++)
        a[i,j]=b[j,i]
```

【図２０ｂ】

FIG. 20b

```
for(i=2,i<(n-1),i++)
  a[i] +=a[1-i]==[i+1]
```

τ ⟹

```
for (i=2,i<(n-2),i+=2) {
  a[i] += n[i-1]=a[i+1];
  a[i+1] += a[i]=a[i+2];
};
if (((n-2) % 2) == 1)
  x[n-1] += a[n-2]=a[n]
```

【図２０ｃ】

FIG. 20c

```
for(i=1,i<n,i++) {
  a[i] += c;
  x[i+1]=d+x[i+1]=a[i]
}
```

τ ⟹

```
for (i=1,i<n,i++)
  a[i] += c;
for (i=1, i<n, i++)
  x[i+i] <d+x[i+1]=a[i]
```

【図２１ａ】

FIG. 21a

g(V) p	g(V) q	f(p,q) V	2p + q
0	0	False	0
0	1	True	1
1	0	True	2
1	1	False	3

【図２１ｂ】

FIG. 21b

VAL[p,q]		p = 0	p = 1
q	0	0	1
	1	1	0

【図２１ｃ】

FIG. 21c

AND[A,B]		A = 0	A = 1	A = 2	A = 3
B	0	3	0	0	0
	1	3	1	2	3
	2	0	2	1	3
	3	3	0	0	3

【図２１ｄ】

FIG. 21d

OR[A,B]		A = 0	A = 1	A = 2	A = 3
B	0	3	1	2	3
	1	1	1	2	2
	2	2	2	1	1
	3	0	1	2	0

【図２１ｅ】

```
(1)  bool A,B,C;                    (1')  short a1,a2,b1,b2,c1,c2;
(2)  A = True;                      (2')  a1=0; a2=1;
(3)  B = False;                     (3')  b1=0; b2=0;
(4)  C = False;                     (4')  c1=1; c2=1;
(5)  C = A & B;          T          (5')  x=AND[2*a1+a2,2*b1+b2]; c1=x/2; c2=x%2;
(6)  C = A & B;                     (6')  c1=(a1 ^ a2) & (b1 ^ b2); c2=0
(7)  C = A | B;                     (7')  x=OR[2*a1+a2,2*b1+b2]; c1=x/2; c2=x%2;
(8)   if (A) ...;                   (8')  x=2*a1+a2; if ((x==1) || (x==2) ...;
(9)   if (B) ...;                   (9')  if (b1 ^ b2) ...;
(10)  if (C) ...;                   (10') if (VAL[c1,c2]) ...;
```

FIG. 21e

【図２２】

```
String G (int n) {
   int i=0,k;
   String B;
   while (i) {
      L1:  if (n==1) {S[i++]="A";k=0;goto L6};
      L2:  if (n==2) {S[i++]="B";k= -2 ;goto L6};
      L3:  if (n==3) {S[i++]="C";goto L8};
      L4:  if (n==4) {S[i++]="K";goto L9};
      L5:  if (n==5) {S[i++]="C";goto L11};
           if (n>12) goto L1;
      L6:  if (k++<=2) {S[i++]="A";goto L6} else goto L8;
      L8:  return S;
      L9:  S[i++]="C"; goto L10;
      L10: S[i++]="B"; goto L8:
      L11: S[i++]="C"; goto L12;
      L12: goto L10;
   }
}
```

FIG. 22

【図２３ａ】

FIG. 23a

$$Z(X + r,Y) = 2^{32} \cdot Y + (r + X) = Z(X,Y) + r$$
$$Z(X,Y + r) = 2^{32} \cdot (Y + r) + X = Z(X,Y) + r \cdot 2^{32}$$
$$Z(X \cdot r,Y) = 2^{32} \cdot Y + X + r = Z(X,Y) + (r - 1) \cdot X$$
$$Z(X,Y \cdot r) = 2^{32} \cdot Y \cdot r + X = Z(X,Y) + (r - 1) \cdot 2^{32} \cdot Y$$

【図２３ｂ】

FIG. 23b

```
(1) int X=45, Y=95;                 (1') long Z=167759066119551045;
(2) X += 5;                  τ      (2') Z += 5;
(3) Y += 11;                        (3') Z += 47244640256;
(4) X *= c;                         (4') z += (c-1)* (Z & 4294967295);
(5) Y *= d;                         (5') Z += (d-1)* (Z & 18446744069414584320);
```

【図２４ａ】

```
(1)  int A[9];                      (1')  int A1[4],A2[4];
(2)  A[i] = ...;                    (2')  if ((i%2)==0) A1[i/2] =...
       ...                                else A2[i/2]=...;
       ...                                 ...
(3)  int B[8],C[19];                (3')  int BC[20];
(4)  B[i] = ...;                    (4')  BC[3*i] = ...;
(5)  C[i] = ...;                    (5')  BC[i/2*3+1+i%2] = ...;
       ...                          (6')  int D1[1,4];
(6)  int D[9]              τ        (7')  for(j=0;j<=1;j++)
(7)  for(i=o;i<=B;i++)                       for(k=0;k<=4;k++)
       D[i]=2 * D[i+1];                        if (k==4)
       ...                                       D1[j,k]=2*D1[j+1,0];
       ...                                     else
       ...                                       D1[j,k]=2*D1[j,k+1];
(8)  int E[2,2];                             ...
(9)  for(i=Q;i<=2;i ++)             (8')  int E1[8]
       for(j=0;i<=2;i++)            (9')  for(i=0;1<=8;i++)
         swap(E(i,j], E[j,i]);              swap(E[i], E[3=(i%3)+i/3]);
```

FIG. 24a

【図２４ｂ】

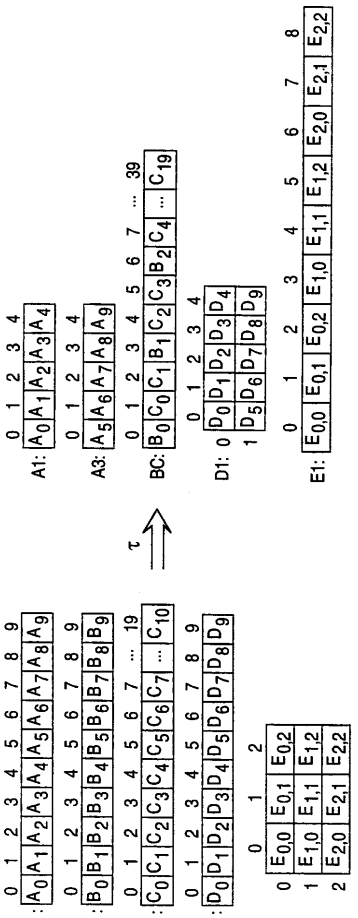


FIG. 24b

【図２５ａ】

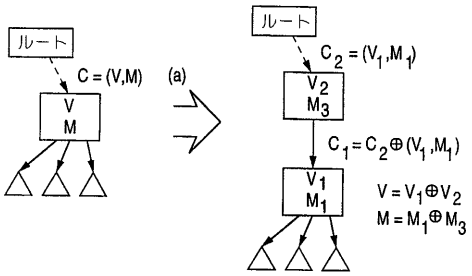


FIG. 25a

【図２５ｂ】

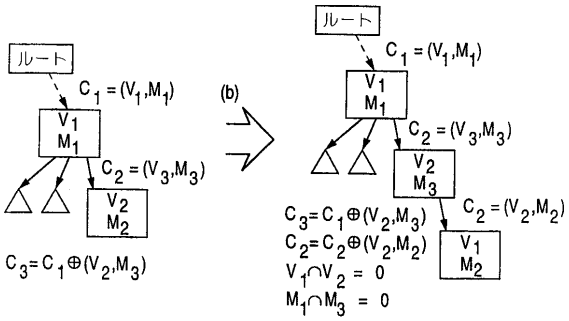


FIG. 25b

【図２５ｃ】

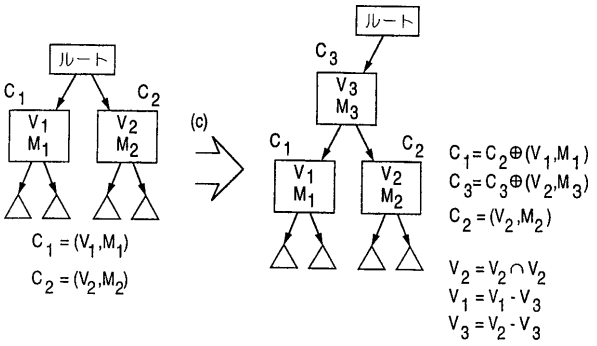


FIG. 25c

【図２５ｄ】

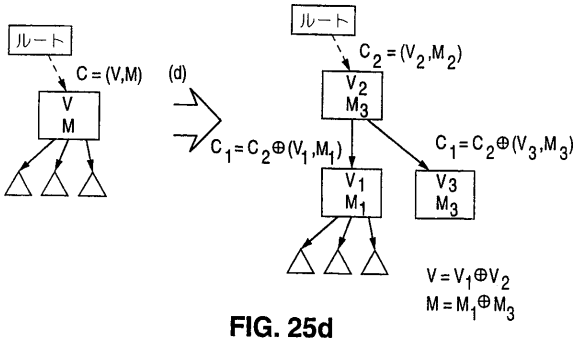
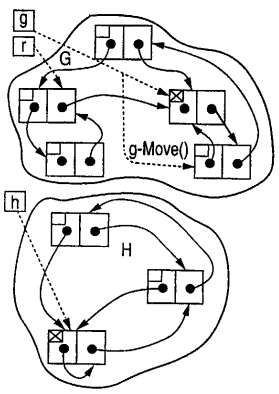


FIG. 25d

【図２６】




```
Node g, h;
method P(...,Node f) {
 /*1*/ g = g.Move();
       h = h.Move();
 /*2*/ h = h.Insert(new Node);
       ⋮
 /*3*/ x.R(...,f.Move());
       ⋮
 /*4*/ if (f==g) ?...
 /*5*/ if (g==h) F...
       ⋮
 /*6*/ f.Token=False;
       g.Token=True;
 /*7*/ if (f.Token)? ...
       ⋮
 /*8*/ f.Token=True;
       h.Token=False;
 /*9*/ if (f.Token) T ...
}
```

FIG. 26

【図２７】

```
thread S {
  int R;
  while (1) {
    R = random(1,C);
    X = R*R;
    sleep(3);
  }
}

thread T {
  int R;
  while (i) {
    R = random(1,C);
    X = 7*R*R;
    sleep(2);
    X *= X;
    sleep(5);
  }
}

int X, Y;
const C = sqrt(maxint)/10;
main () {
  S.run(); T.run();
    ...
  if ((Y - 1) == X) F <= P
    ...
}
```

FIG. 27

【図２８】

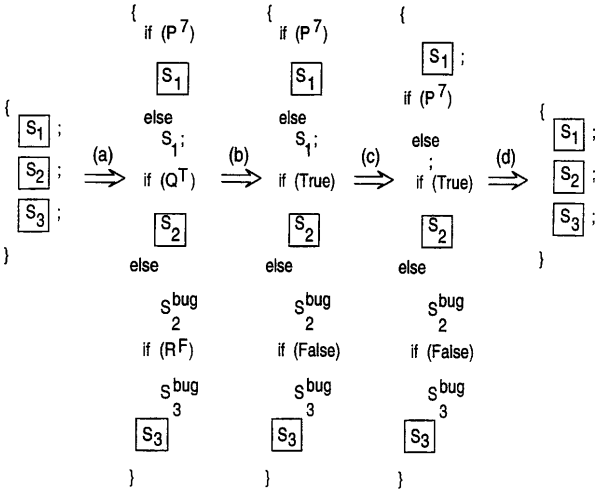


FIG. 28

【図３０】

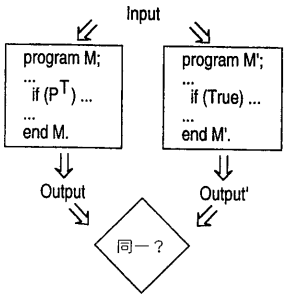


FIG. 30

【図３１ａ－２】

スカラーからオブジェクトへプロンプトする

混乱化 目標	操作	変換	品質 効力	弾力性	コスト	メトリクス	セクション
データ	記憶&符号化	符号化の変更	符号化関数の複雑性に依存する			μ1	7.1.1
			低	強	無料		7.1.2
		可変寿命の変更	低	強	無料	μ4	7.1.2
		変数の分割	原変数が分割される変数の数に依存する			μ1	7.1.3
		静的データから手続データへの転換	生成関数の複雑性に依存する			μ1,μ2	7.1.4
	集合	併合スカラー変数	低	弱	無料	μ1	7.2.1
		ファクタクラス	中	†	無料	μ1, μ7b,c,e	7.2.3
		挿入Bogus クラス	中	†	無料	μ1, μ7b,c	7.2.3
		再ファクタクラス	中	†	無料	μ1, μ7b,c,e	7.2.3
		分割アレイ	†	弱	無料	μ1, μ2, μ6	7.2.2
		併合アレイ	†	弱	無料	μ1, μ3	7.2.2
		フォールドアレイ	†	弱	安価	μ1,μ2,μ3,μ6	7.2.2
		平担化アレイ	†	弱	無料		7.2.2
	オーダリング	再オーダーメソッド&インスタンス変数	低	ワンウェイ	無料		7.3
		再オーダーアレイ	低	弱	無料		7.3

FIG. 31a-2

【図３１ｂ】

混乱化 目標	操作	変換	品質 効力	弾力性	コスト	メトリクス	セクション
目標となる	予防	HoseMocha	低	トリビアル	無料	μ1	9
	本来	スライシングを予防するために別名化形式を付加する	中	強	無料	μ1,μ5	9.4
		スライシングを予防するために変数従属性を付加する	不明瞭述語の品質に依存する			μ1	9.4
		Bogus データ従属性を付加する	中	弱	安価	μ1	9.1.1
		サイド効果を有する不明瞭な述語を使用する	中	弱	無料	μ1	9.5
		ティフィカルト定理を用いて不明瞭な述語を作る	†	†	†	μ1	9.5

FIG. 31b

【図２９】

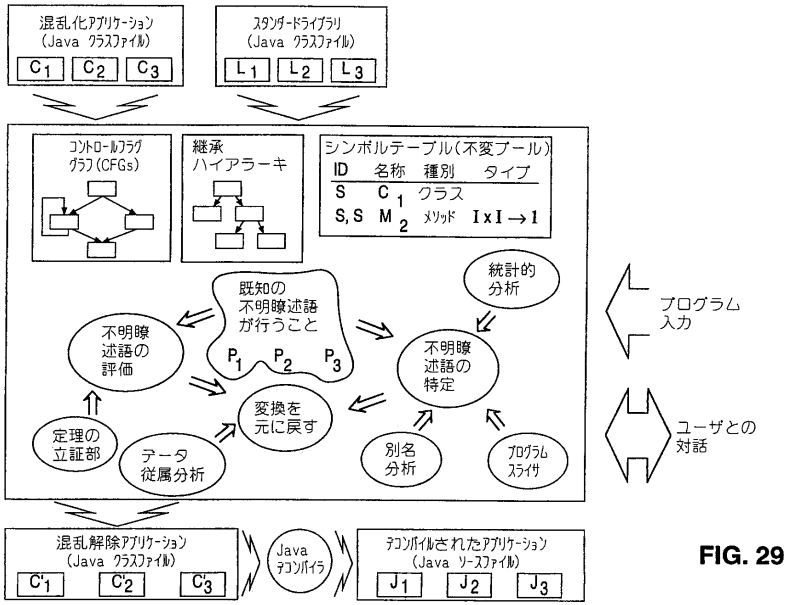


FIG. 29

【図３１ａ－１】

混乱化 目標	操作	変換	品質 効力	弾力性	コスト	メトリクス	セクション
レイアウト		IDのスクランブル	中	ワンウェイ	無料		5.5
		フォーマッティングの変更	低	ワンウェイ	無料		5.5
		コメントの除去	高	ワンウェイ	無料		5.5
コントロール	計算	デッド又はイレリバントコードの挿入	不明瞭述語の品質と、コンストラクトが挿入される入れ子深さとに依存する			μ1, μ2, μ3	6.2.1
		ループ条件の拡張				μ1, μ2, μ3	6.2.2
		可約から非可約へ				μ1, μ2, μ3	6.2.3
		冗長オペランドの付加				μ1	6.2.6
		プログラム用イディオムの除去	中	強	†	μ1	6.2.4
		テーブル解釈	高	強	高価	μ1	6.2.5
	集合	インラインメソッド	中	ワンウェイ	無料	μ1	6.3.1
		アウトライン文	中	強	無料	μ1	6.3.1
		インタリーブメソッド	不明瞭述語の品質に依存する			μ1, μ2, μ3	6.3.2
		クローンメソッド				μ1, μ7	6.3.3
		ブロックループ	低	弱	無料	μ1, μ2	6.3.4
		アンロールループ	低	弱	安価	μ1	6.3.4
		ループ分裂	低	弱	無料	μ1, μ2	6.3.4
	オーダリング	再オーダー文	低	ワンウェイ	無料		6.4
		再オーダーループ	低	ワンウェイ	無料		6.4
		再オーダー式	低	ワンウェイ	無料		6.4

FIG. 31a-1

【図３２】

不明瞭構造	弾力性	品質 コスト	セクション
ライブラリ関数の呼び出しから生成される	トリビアル	ライブラリ関数のコストに依存する	6.1.1
ローカル（内部基本ブロック）情報から生成される	トリビアル	無料...安価	6.1.1
グローバル（内部基本ブロック）情報から生成される	弱	無料...安価	6.1.1
内部手続並びに別名情報から生成される	フル	安価...高価	8.1
プロセス相互作用並びにスケジューリングから生成される	フル	安価...高価	8.2

FIG. 32

フロントページの続き

(74)代理人
弁理士 樋口 外治
(72)発明者 コールベルウ，クリスチャン スベン
ニュージーランド国，オークランド，マウント エデン，グレナルモンド ロード ２５
(72)発明者 トムボーソン，クラーク デビット
ニュージーランド国，オークランド，メドウバンクス，ファンコート ストリート ３／６１
(72)発明者 ロウ，ダグラス ウェイ コック
ニュージーランド国，オークランド ３，エプソム，アルモラー ロード ５６

審査官 市川 武宜

(56)参考文献 門田 暁人 AKito MONDEN，プログラムの難読化法の実験的評価 An Experiment to Evaluate Methods for Program Scrambling，情報処理学会研究報告 Ｖｏｌ．９６ Ｎｏ．３２ IPSJ SIG Notes，日本，社団法人情報処理学会 Information Processing Society of Japan，１９９６年 3月２２日，第96巻，３３－４０頁
村山 隆徳 Takanori MURAYAMA，ソフトウェアの難読化について How to make a software program hard to understand，電子情報通信学会技術研究報告 Ｖｏｌ．９５ Ｎｏ．３５３ IEICE Technical Report，日本，社団法人電子情報通信学会 The Institute of Electronics,Information and Communication Engineers，１９９５年１１月１６日，第95巻，９－１４頁
Robert S. Swanke，他人の作ったプログラムの上手な解析方法 The Art of Reverse Engineering，Ｃ ＭＡＧＡＺＩＮＥ 第３巻 第１０号，日本，ソフトバンク株式会社，１９９１年１０月 １日，第３巻，４２－４４頁

(58)調査した分野(Int.Cl.，ＤＢ名)
G06F 21/22