(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号
特許第5542142号
(P5542142)
(45)発行日　平成26年7月9日(2014.7.9)　(24)登録日　平成26年5月16日(2014.5.16)

(51)Int.Cl.　F I
G06F 9/44 (2006.01)　G06F 9/06 620K
G06F 9/06 620A

請求項の数 19　(全 42 頁)

(21)出願番号　特願2011-530204 (P2011-530204)
(86)(22)出願日　平成21年9月30日 (2009.9.30)
(65)公表番号　特表2012-504823 (P2012-504823A)
(43)公表日　平成24年2月23日 (2012.2.23)
(86)国際出願番号　PCT/US2009/059112
(87)国際公開番号　WO2010/039893
(87)国際公開日　平成22年4月8日 (2010.4.8)
審査請求日　平成24年9月18日 (2012.9.18)
(31)優先権主張番号　12/244,959
(32)優先日　平成20年10月3日 (2008.10.3)
(33)優先権主張国　米国 (US)

(73)特許権者　500046438
マイクロソフト　コーポレーション
アメリカ合衆国　ワシントン州　９８０５２－６３９９　レッドモンド　ワン　マイクロソフト　ウェイ
(74)代理人　100140109
弁理士　小野　新次郎
(74)代理人　100075270
弁理士　小林　泰
(74)代理人　100101373
弁理士　竹内　茂雄
(74)代理人　100118902
弁理士　山本　修
(74)代理人　100153028
弁理士　上田　忠

最終頁に続く

(54)【発明の名称】宣言型プログラミング言語の木ベースの有向グラフのプログラミング構造

(57)【特許請求の範囲】
【請求項１】
宣言型プログラミング言語のプログラミングコンストラクトを表すコンピューターにより実施可能な方法であって、
第１の構文木データ構造を表す宣言型プログラミング言語および第２の構文木データ構造を表す宣言型プログラミング言語を受信するステップと、
複数の構文スコープレゾリューションの代替案の中から１つの構文スコープレゾリューションの代替案の選択の指示を受信するステップであって、前記複数の構文スコープレゾリューションの代替案のそれぞれは、前記第１の構文木データ構造から前記第２の構文木データ構造へのクロスリファレンスの異なる方法を定義する、ステップと、
前記第１の構文木データ構造のノードおよび前記第２の構文木データ構造のノードの間の少なくとも１つのクロスリファレンスと少なくとも２つのファクタリングされた定義とを含んでいる有向グラフデータ構造を作成するステップであって、前記少なくとも１つのクロスリファレンスは、前記選択された構文スコープレゾリューションの代替案に基づく、ステップと、
前記少なくとも２つのファクタリングされた定義を、前記少なくとも２つのファクタリングされた定義と等価の前記有向グラフデータ構造の組み合わせ定義に自動的に組み合わせるステップと
を含むことを特徴とする方法。
【請求項２】

前記構文スコープレゾリューションの代替案の選択の指示は、上位レベルのノードから始まり１または複数の下位レベルのノードに進む、前記有向グラフデータ構造を作成するステップの間の、構文スコープを解決すべきことを示す指示を含むことを特徴とする請求項１に記載の方法。

【請求項３】

前記構文スコープレゾリューションの代替案の選択の指示は、内部レベルのノードから始まり１または複数の外部レベルのノードに進む、前記有向グラフデータ構造を作成するステップの間の、構文スコープを解決すべきことを示す指示を含むことを特徴とする請求項１に記載の方法。

【請求項４】

前記構文スコープレゾリューションの代替案の選択の指示は、現在のノードから始まる、前記有向グラフデータ構造を作成するステップの間の、構文スコープを解決すべきことを示す指示を含むことを特徴とする請求項１に記載の方法。

【請求項５】

前記構文スコープレゾリューションの代替案の選択の指示は、現在のノードの後続処理のノードから始まる、前記有向グラフデータ構造を作成するステップの間の、構文スコープを解決すべきことを示す指示を含むことを特徴とする請求項１に記載の方法。

【請求項６】

前記有向グラフデータ構造の少なくとも１つの定義を、組み合わさって、当該少なくとも１つの定義と等価である少なくとも２つの下位の定義にファクタリングするステップをさらに備えたことを特徴とする請求項１記載の方法。

【請求項７】

少なくとも２つの異なるファイルに前記少なくとも２つの下位の定義の値を格納するステップをさらに備えたことを特徴とする請求項６記載の方法。

【請求項８】

前記構文スコープレゾリューションの代替案の選択の指示は、ノードのラベルに先行する「．」を含み、前記「．」は、構文スコープを上位から下位の方法で解決すべきであることを示すことを特徴とする請求項１に記載の方法。

【請求項９】

前記構文スコープレゾリューションの代替案の選択の指示は、ノードのラベルに先行する「＆」を含み、前記「＆」は、構文スコープを現在のノード、または前記現在のノードの後続処理のノードから始めて解決すべきであることを示すことを特徴とする請求項８に記載の方法。

【請求項１０】

前記構文スコープレゾリューションの代替案の選択の指示は、ノードのラベルに先行する「＆」を含み、前記「＆」は、構文スコープを現在のノード、または前記現在のノードの後続処理のノードから始めて解決すべきであることを示すことを特徴とする請求項１に記載の方法。

【請求項１１】

宣言型プログラミングモデルのプログラミングコンストラクトを表すコンピューターにより実施可能な方法であって、

複数の構文スコープレゾリューションの代替案の中から１つの構文スコープレゾリューションの代替案を選択することを含む処理を通じて作成される有向グラフデータ構造を受信するステップであって、

該有向グラフデータ構造が、少なくとも共有親ノード、第１の後続処理のノード、および第２の後続処理のノードを含むノードのセット、アークのセット、並びに少なくとも第１の定義および第２の定義を含む前記宣言型プログラミングモデルのプログラミングコンストラクトの定義のセットを含み、

前記複数の構文スコープレゾリューションの代替案のそれぞれは、前記第１の構文木データ構造から前記第２の構文木データ構造へのクロスリファレンスの異なる方法を定義し

、
前記第１の後続処理のノードを定義する前記第１の定義を受信するステップであって、前記第１の後続処理のノードは前記共有親ノードの後続処理である、ステップと、
前記第２の後続処理のノードを定義する前記第２の定義を別個に受信するステップであって、前記第２の後続処理のノードは前記共有親ノードの後続処理である、ステップと
を含む、前記有向グラフデータ構造を受信するステップと、
前記第１の定義および前記第２の定義を単一の複合定義に組み合わせるステップであって、前記単一の複合定義は、前記第１の定義および前記第２の定義の組み合わせに論理的に等価である、ステップと
を含むことを特徴とする方法。
【請求項１２】
前記有向グラフデータ構造を受信するステップは、前記有向グラフデータ構造を作成するステップを含むことを特徴とする請求項１１記載の方法。
【請求項１３】
前記定義のセットのうちの別の定義を、合わせると、当該別の定義と等価である少なくとも２つのファクタリングされた定義に分けるステップをさらに備えたことを特徴とする請求項１１記載の方法。
【請求項１４】
前記有向グラフデータ構造は、前記定義のセットの定義を記述するラベルをさらに含み、前記組み合わせるステップは、共通のラベルを共有する複数の定義の上位から下位への結合を行うステップを含むことを特徴とする請求項１１記載の方法。
【請求項１５】
前記組み合わせるステップは、前記共通のラベルの後続処理の上位から下位への結合を行うステップを含むことを特徴とする請求項１１記載の方法。
【請求項１６】
少なくとも２つの異なるファイルに前記少なくとも２つのファクタリングされた定義を格納するステップをさらに備えたことを特徴とする請求項１３記載の方法。
【請求項１７】
有向グラフデータ構造を分析するためのコンピューター実行可能命令を含むコンピューター可読記録媒体であって、
コンストレイントベースの型の定義をサポートし順序無関係の実行モデルに従う宣言型プログラミングモデルを表している有向グラフデータ構造を作成するかまたは受信するＤグラフモジュールであって、
前記有向グラフデータ構造を作成する場合、前記Ｄグラフモジュールは、複数の構文スコープレゾリューションの代替案の中から、前記有向グラフデータ構造を作成する際に利用すべき、１つの構文スコープレゾリューションの代替案を決定し、
複数の構文スコープレゾリューションの代替案のそれぞれは、第１の構文木データ構造から前記第２の構文木データ構造へのクロスリファレンスの異なる方法を定義し、
前記有向グラフデータ構造は少なくとも２つのファクタリングされた定義を含み、
前記少なくとも２つのファクタリングされた定義は、前記少なくとも２つのファクタリングされた定義と等価の前記有向グラフデータ構造の組み合わせ定義に自動的に組み合わさる、
Ｄグラフモジュールと、
ファクタリングすることを適用することができる前記有向グラフデータ構造のうちの少なくとも１つの定義を識別して、前記少なくとも１つの定義を表している少なくとも１つの等価の定義を形成するファクタリングモジュールと
を備える、コンピューター可読記録媒体。
【請求項１８】
前記ファクタリングモジュールは、前記少なくとも１つの定義を自動的にファクタリングして、前記少なくとも１つの等価の定義を含む等価の有向グラフデータ構造を形成する

ことを特徴とする請求項１７記載のコンピューター可読記録媒体。
【請求項１９】
前記等価の有向グラフデータ構造を、ストレージまたは第三パーティのアプリケーションのうち少なくとも１つに出力する出力モジュールをさらに備えたことを特徴とする請求項１８記載のコンピューター可読記録媒体。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は、一般に、宣言型プログラミング言語の木ベースの有向グラフのプログラミング構造に関する。
【背景技術】
【０００２】
一般的な背景として、コンピューターサイエンスにおいて、抽象構文木（ＡＳＴ）は、プログラミング言語で書かれたあるソースコードの構文を表す木表現またはソースコードを表す他の何らかの機能的に等価の表現である。木の各ノードは、ソースコードで生じるコンストラクトを示している。木は、元のソースに現れるいくつかのコンストラクトを表すことができないという点で抽象的である。かかる省略の例には、挿入語句をグループ化することがある。ＡＳＴでは、オペランドのグループ化は木構造内に潜在しているからである。
【０００３】
ＡＳＴは、ソースコードをコンパイルするプロセスの一部として、パーサーによって構築されることが多い。一旦構築されると、追加の情報は、意味解析などの次の処理によってＡＳＴに追加される。意味解析はＡＳＴに基づく抽象意味グラフ（ＡＳＧ）を作成する結果になり得る。ＡＳＧは、ＡＳＴより上位の抽象概念であり、ＡＳＴは表現またはプログラムの統語構造を表すために用いられる。コンピューターサイエンスにおいて、ＡＳＧは、プログラミング言語などの形式的言語において表現の意味を表すかまたは表現を抽出する際に用いられるデータ構造である。
【０００４】
ＡＳＧは通常、拡充して抽象化するプロセスによって抽象構文木から構築される。例えば、拡充は、識別子ノードからの逆ポインタまたはエッジを追加することであってもよい。識別子ノードにおいて、変数は、その変数の宣言を表しているノードに対して用いられている。例えば、抽象化は、意味のためではなく、構文解析においてだけ関係のある詳細を除去する必要がある。
【０００５】
この点に関して、ＸＭＬなどの半構造化データについての現在の表現は、木構造を表すことに限られる。ＸＭＬについては、グラフ構造を表すことは、ＸＭＬ＿ＩＤなどの、明示的な参照を用いることを必要とする。このことは、基本的なグラフ構造の表現及び格納に関して複雑化させかつ柔軟性に欠ける。例えば、ＸＭＬ　ＩＤの使用は、識別子が何であるかかつ参照が何であるかを型システムが定義することを必要とする。これは、使用の難しさをもたらす基本的なグラフ構造に対して外部であることを意味する。
【０００６】
従って、明示的な識別子を用いずに、コンパクトで使いやすい構文を用いる複雑なグラフ構造化データを書く能力についての未解決の必要性が存在する。グラフとしての半構造化プログラムデータの現在の表現についての上記した不足は、従来のシステムについてのいくつかの問題の概要を提供することを単に意図しており、網羅的であることを意図していない。従来のシステムに関する他の問題及び本明細書において説明される様々な非限定的な実施形態の対応する利点は、以下の詳細な説明を検討することで更に明らかになるだろう。
【発明の概要】
【０００７】

簡略化した概要が本明細書において提供されて、更なる詳細な説明及び添付の図面において続く例示的で非限定な実施形態の様々な態様の基本的理解または一般的な理解を可能にするのを助ける。しかしながら、この概要は、広範囲に及んで概説することまたは網羅的に概説することを意図していない。代わりに、この概要の唯一の目的は、続く様々な実施形態の更に詳細な説明の導入部として、簡略化した形式でいくつかの例示的で非限定的な実施形態に関するいくつかの概念を示すことである。

【０００８】

宣言型プログラミング言語についてのツリーベースの有向グラフのプログラミング構造の実施形態が提供される。様々な実施形態において、本明細書の１つの非限定的な実施形態において「ＤＧｇｒａｐｈｓ」と称される、複雑なグラフ構造化データは、明示的な識別子を用いることなしに、コンパクトで使いやすい構文を用いて書かれる。１つの非限定的な態様において、構文は、ファクタリングされた関係とも称される、適合関係のサポートを含む。別の非限定的な態様では、半構造化されたグラフデータは、木ベースの表現であり、構文は、参照の語彙的解明または構文スコーピング及び／または非ローカルの初期化を含む。

【０００９】

これらの実施形態及び他の実施形態は、更に詳細に以下で説明される。

【００１０】

様々な非限定的な実施形態は、添付の図面を参照して更に説明される。

【図面の簡単な説明】

【００１１】

【図１】宣言型プログラミング言語及び関連する構造についての一連のコンパイル処理のブロック図である。

【図２】適合関係の一般的な説明を示す図である。

【図３】有向グラフ構造についての第１の非限定的な態様を示すフローダイアグラムの図である。

【図４】有向グラフ構造についての第２の非限定的な態様を示すフローダイアグラムの図である。

【図５】本明細書において説明される１つまたは複数の態様に基づくコンピューター実行可能モジュールを有するコンピューター可読媒体として例示的な実施形態を示すブロック図である。

【図６】説明のための半構造化されたグラフデータのサンプル表現の図である。

【図７】説明のための半構造化されたグラフデータのサンプル表現の図である。

【図８】説明のための半構造化されたグラフデータのサンプル表現の図である。

【図９】１つまたは複数の実施形態において説明されるように、木ベースの構造と関連して、構文スコーピングの例示的な態様を示す図である。

【図１０】選択の比較を可能にする有向グラフ構造についての異なる構文スコーピングの選択についてのテキスト表現への動的マッピングを示す図である。

【図１１】選択の比較を可能にする有向グラフ構造についての異なる構文スコーピングの選択についてのテキスト表現への動的マッピングを示す図である。

【図１２】選択の比較を可能にする有向グラフ構造についての異なる構文スコーピングの選択についてのテキスト表現への動的マッピングを示す図である。

【図１３】選択の比較を可能にする有向グラフ構造についての異なる構文スコーピングの選択についてのテキスト表現への動的マッピングを示す図である。

【図１４】一連の類似または関連する有向グラフ構造の図を示して、グラフ構造を説明するためのコンパクトで柔軟性のある構文についての追加の態様を示す図である。

【図１５】一連の類似または関連する有向グラフ構造の図を示して、グラフ構造を説明するためのコンパクトで柔軟性のある構文についての追加の態様を示す図である。

【図１６】一連の類似または関連する有向グラフ構造の図を示して、グラフ構造を説明するためのコンパクトで柔軟性のある構文についての追加の態様を示す図である。

【図１７】一連の類似または関連する有向グラフ構造の図を示して、グラフ構造を説明するためのコンパクトで柔軟性のある構文についての追加の態様を示す図である。
【図１８】一連の類似または関連する有向グラフ構造の図を示して、グラフ構造を説明するためのコンパクトで柔軟性のある構文についての追加の態様を示す図である。
【図１９】宣言型プログラミングモデル及び関連する表現についての例示的なプロセスチェーンの図である。
【図２０】記録指向型の実行モデルに付随する型システムの図である。
【図２１】本発明の実施形態に基づくコンストレイントベースの実行モデルに付随する型システムの非限定的な図である。
【図２２】順序実行モデルに基づくデータ格納の図である。
【図２３】順序に依存しない実行モデルに基づくデータ格納の非限定的な図である。
【図２４】本明細書において説明されている様々な実施形態を実施することができる例示的で非限定的なネットワーク化された環境を表すブロック図である。
【図２５】本明細書において説明されている様々な実施形態のうちの１つまたは複数の態様を実施することができる例示的で非限定的なコンピューターシステム環境または動作環境を表すブロック図である。
【発明を実施するための形態】
【００１２】
背景技術において検討したように、特に、従来のシステムは、コンパクトで使いやすい構文を用いて複雑なグラフ構造化データを書くことを可能にしていない。複雑なグラフ構造化データはＸＭＬを用いて表現されるが、表現は、ＸＭＬ＿ＩＤのような明示的な識別子または参照を用いて、構造化グラフデータを形成しなければならない。
【００１３】
従って、様々な非限定的な実施形態において、宣言型プログラミング言語のための木ベースの有向グラフのプログラミング構造についての実施形態が提供される。本明細書のいくつかの実施形態において「Ｄグラフ」と称される、複雑なグラフの構造化データは、明示的な識別子を用いることなくコンパクトで使いやすい構文を用いて書くことができる。様々な非限定的な態様において、構文は、適合関係へのサポートを含み、参照の語彙的な解明（別名、構文スコーピング）及び／または非ローカルの初期化を可能にする。
【００１４】
その後に続くことのロードマップとして、様々な実施形態の概説が最初に提供され、次に例示的で非限定的な任意の実施例並びに例示的なグラフ及びグラフ表現が、更なる理解のために更に詳細に検討される。次に、いくつかの補足的な内容が、Ｄプログラミング言語などの宣言型プログラミング言語と関連して提供される。最後に、本明細書において説明されている技術を配備することができるいくつかの例示的で非限定的なネットワーク環境及びコンピューティング環境が記載されている。
【００１５】
一般に、宣言型プログラミング言語においてソースコードを書くことが望ましい場合が多く、命令型プログラミング言語に対応するものと考えられる場合が多い。命令型プログラミング言語と異なって、宣言型プログラミング言語は、ユーザに、自分たちの所望が所定の技術またはプラットフォームに対して満足させる方法を特定する必要なしに自分たちのデータから自分たちが消耗するものを書き出せるようにする。この点について、その更なる詳細を以下に見出すことができるＤプログラミング言語（即ち、略して「Ｄ」）は、コンパクトで人間の理解できる表現に適していてかつ、フラッシュストレージ、リレーショナルデータベース、ＲＡＭ、外付けドライブ、ネットワークドライブ等であれ基本的な格納メカニズムと無関係のデータ集約型のアプリケーションを作成して修正する効率コンストラクトを含むことが有利な宣言型プログラミング言語である。「Ｄ」はまた、「Ｍ」プログラム言語と称される場合があるが、整合性を図るために「Ｍ」という言い回しは本明細書において用いられていない。
【００１６】

Ｄは、宣言型ソースコードを書き込むための効率的でコンパクトな構文を含む。この点について、Ｄのプログラミングコンストラクトはまた、コンパイラが受信する所定のソースコードについて生成される１つまたは複数の抽象構文木に基づいて半構造化されたグラフデータとして効率的に表すことができる。更に、半構造化されたグラフデータについての構文の使いやすい性質のために、アプリケーション及び開発者は、半構造化されたグラフデータを直接特定することができる場合に、実際のソースコードを形成する必要はない。この点について、木のセットまたはＤプログラムの別の仕様に基づいて、より簡単なＤグラフ構造の形成に寄与する構文においてサポートされる様々な新規の特定のためにＤプログラムのプログラミングコンストラクトを効率的に表すＤグラフを形成することができる。Ｄグラフ構造は、マシンに理解可能であってかつ意味グラフの従来のテキスト表現と同様に難解でないＤグラフ構造のテキスト表現の目視検査に基づいて人間にも理解可能である。

【００１７】

Ｄプログラムを表わして用いることができる異なる方法を示す一般的なブロック図システムが、図１の一連のコンパイルに示されている。例えば、ソースコード１００を、開発者またはマシンが直接書くことができる。ソースコード１００は、Ｄコンパイラ１１０がコンパイルすることができる。Ｄコンパイラ１１０は、例えばソースコード１００を解析するためのＤパーサー１２０及びＤ構造木を形成するためのＤ構造木コンポーネント１３０を含み、次にＤ構造木は解析されてＤグラフ構造１４０に変換される。Ｄグラフ構造１４０は、開発者が、またアプリケーションが直接生成して、Ｄグラフ構造でがサポートするいくつかの特徴に基づいて、コンパクトな方法で表すことができる。Ｄグラフ構造１４０は、木１３０へほどくことができて、ソースコード１００に戻すことができる。Ｄグラフ構造１４０は、様々なドメイン特定使用１５０に基づいてデータ集約アプリケーションに関連してオブジェクトコードにコンパイルすることができるかまたはセミコンパイルすることができる。ドメイン特定使用は、例えばＳＱＬデータベースクエリ、企業データ管理（ＥＤＭ）、スプレッドシートアプリケーション、グラフィックスアプリケーション、即ちデータを格納して分析することができるどこでもである。

【００１８】

この点について、以前に実現されていなかったＤグラフ構造１４０の２つの特徴は、適合関係へのサポート及び識別子または参照についての構文レゾリューションを含む。

【００１９】

一般的に言えば、図２に示すように、適合関係２２０は、１つのモデル２１０をその参照モデル２００と称される別のモデルにリンクさせる。より具体的には、本明細書において説明されている宣言型プログラミングモデルの内容における適合関係２２０は、共通のラベルのためのファクタリングされた定義を特定する可能性及び、有向グラフ構造の効率的な表現のためにこれらを結合することを意味する。

【００２０】

例えば、図３は通常、１つの実施形態において有向グラフとして宣言型プログラミングモデルのプログラミングコンストラクトを表す方法を示す。３００において、有向グラフのデータ構造は、宣言型プログラミングモデルのプログラムコンストラクトの定義のセットを含むよう受信される。３１０において、ファクタリングが定義のセットのうちのいずれかにあてはまるかどうかの決定が行われる。あてはまる場合、３２０において、定義は原理をファクタリングすることに基づいて共通の等価の定義を形成することができるよう組み合わせることができるかまたは、定義は格納に便利という理由かまたは他の理由で別々に分ける（ファクタリングする）ことができる。ファクタリングは、受信されるかまたは格納から検索される有向グラフ構造に適用することができるかまたは、有向グラフ構造が作成される時に適用することができる。

【００２１】

動的スコーピングとは対照的に、構文レゾリューションまたは構文スコーピングは、柔軟性がありかつ任意の有向グラフを作るためにＤプログラミング言語に関連することが有

利である。例えば、図４は一般に、木表現に基づく有向グラフとして宣言型プログラミングモデルのプログラミングコンストラクトを表す方法を示す。４００において、宣言型プログラミング言語についての構文木が受信される。４１０において、１つの構文木を相互参照する異なる方法を定義している構文スコープの指示が特定され、次に４２０において、有向グラフデータ構造は、構文スコープの少なくとも１つの指示に基づいて決定される２つの構文木から、少なくとも１つの相互参照を含むよう生成される。構文スコーピングは、上のレベルから下のレベルまでのスコーピングを適用するかどうかまたは、内側のレベルから外側のレベルまでのスコーピングを適用するかどうかを定義することを含み、現在のノードまたは後続ノードがグラフの所定のエッジにあてはまるかどうかを示すことを含む。

【００２２】

図５に一般的に示されるような１つの実施形態において、コンピューター可読媒体などのコンピュータープログラム製品５００は、宣言型プログラムモデルを表す有向グラフデータ構造を生成または受信するためのＤグラフモジュール５１０を含むことができる。１つの実施形態において、宣言型プログラミングモデルは、コンストレイントベースの型の定義をサポートし、順序無関係の実行モデルに従う。第１の非限定的な態様において説明したように、ファクタリングモジュール５２０が、ファクタリングを適用することができる有向グラフデータ構造の定義を特定するために含まれ得る。第２の非限定的な態様では、有向グラフデータ構造を生成するときに、Ｄグラフモジュールは、有向グラフデータ構造の形成に適用する構文スコープを決定する。

【００２３】

（木ベースの有向グラフプログラミング構造）

上記のように、様々な非限定的な実施形態において、木ベースの有向グラフプログラミング構造が、宣言型プログラミング言語に提供される。いくつかの付加的な背景について、基本的に、Ｄグラフは以下の項目を有するラベル付きの有向グラフである。

【００２４】

１．ノードのセットＮ。

【００２５】

２．Ｎに関する２項関係Ａ。Ａは、有向グラフのアークのセットと称される。よって、アークは、複数のノード対である。

【００２６】

３．ラベルのセットｌ。

【００２７】

４．Ｎｘｌに関する２項関係Ｌ。Ｌは有向グラフのラベルと称される。

図６は、図６００として表される例示的なグラフである。上記の形式を用いて、グラフの図６００は、以下のようなテキストの形で表すことができる。

N={1,2,3,4,5,6,7,8,9,10,11,12,13}

A ={(l,2),(l,3),(2,4),(2,5),(2,6),(3,7),(3,8),(3,9),(4,10),(5,ll),(6,2),(7,l),(8,12),(9,13)}

l = {People, Jack, Jill, Name, Age, Spouse, “Ｊａｃｋ”, 23, “Ｊｉｌｌ”, 25}

L = {(People,l), (Jack,2), (Jill,3), (Name,4), (Age,5), (Spouse,6), (Spouse,7), (Name,8), (Age,9), (“Ｊａｃｋ”,10), (23,ll), (“Ｊｉｌｌ”,12), (25,13)}

このテキスト形式は、全てのグラフを表すのに正確かつ十分であるが、改善される余地がだいぶある。

【００２８】

１．グラフの構造は、図においての場合と同様に明白でない。

【００２９】

２．構文は、完全なグラフを保証しない。例えば、（1,14）は有効な構文であるが、１４はノードでないので有効なエッジではない。

【００３０】

３．テキストは、著者が作成しなければならないグラフ、具体的にはノード及びラベルＩＤにとって重要でない情報を有する。

【００３１】

更に、

４．大きいグラフを書くことは、構成することができる別個のコンポーネントに記述をファクタリングすることによって容易にされる。

【００３２】

同じグラフの以下のテキスト表現を検討する。

```
People {
      Jack{
            Name = “Ｊａｃｋ”,
            Age = 23,
            Spouse = &Jill
      } ,
      Jill {
            Name = “Ｊｉｌｌ”,
            Age = 25,
            Spouse = &Jack
      }
}
```

更なる説明がなくても、この形式のグラフの構造のコンパクトさ及び明白な定義が視覚的に観察される。この点について、Ｄグラフの構文は、更に詳細にかつ以下の一般化された構文を参照して説明される。

```
DGraph:
    Empty
    {DGraph}
    DGraph, DNode
DNode:
    Literal
    Reference
    Identifier = DNode
    Identifier {DGraph}
Reference:
    Path
   .Path
   & Path
   & .Path
Path:
   Identifer
   Path .Identifer
```

構文は、上記のテキスト形式でグラフを書くのを容易にするブロック構造化プログラミングに基づいている。構文識別子{DGraph}は、ラベル及びブロックを有する。識別子は、識別子を用いてラベルを付けられるグラフ内の新規なノードを表す。ブロックDGraphは、その新規なノードの後続処理（ｓｕｃｃｅｓｓｏｒ）を含む。以下のより小さな例は、４つのノード及び３つのエッジを有するグラフを表す。

```
Colors {“Ｒｅｄ”,  ”Ｂｌｕｅ”,  ”Ｙｅｌｌｏｗ”}
```

図７の図７００で示すように、Colorsとラベルを付けられたノードは、“Ｒｅｄ”, ”Ｂｌｕｅ”, ”Ｙｅｌｌｏｗ”とラベルを付けられたノードへの３つエッジを有している。

【００３３】

構文Identifier = DNodeは、正確に
Identifier {DNode}
に等しく、これは単一の後続処理を有するノードである。以下の例は、２つのノード及び１つのエッジを有するグラフを表す。
Age = 23
Ageとラベルを付けられたノードは、23とラベルを付けられたノードへのエッジを有している。以下の構文は、正確に同じグラフを表す。
Age {23}
参照は、Ｄグラフの特徴である。これらを用いることなしに、Ｄグラフは、ＸＭＬ、ＪＳＯＮ（JavaScript Object Notation）、Ｓ式で等を用いて表すことができる木に酷似した木である。一般に、Ｄグラフの参照は、参照がローカルまたは絶対であってもよく、ノード（lvalue）またはその後続処理（rvalue）を検索してもよく、パスを用いて機能を与えられてもよいという点で、非常に豊富である。

【００３４】

Ｄグラフを用いて、主要なアンパサンドを用いた参照は、参照が分解するノードまたは複数のノードへ分解する。アンパサンドは、Ｃ言語におけるIvalueに類似する。アンパサンドなしに、参照は、ラベルによって分解されるノードの後続処理または複数の後続処理に結びつける。図８の図８００を参照すると、以下のは、２つの参照&Jill及びNameを有する。&Jillの参照は、ノードのJill{}にバインドする。Nameの参照は、Nameの後続処理「Jack」にバインドする。
People{Jack{Name{“Ｊａｃｋ”},Age{23},Spouse{&Jill},Nickname=Name},Jill{}}
更に、主要なドット（例えば、ピリオド）を有する参照は絶対であって、上位のレベルのスコープに一致する。主要なドットのない参照は、ローカルであって、ラベルの最内部の発現に一致する。

【００３５】

アンパサンドを用いてこのオプションを構成することは、図９において要約されている４つの選択肢を生じさせて、図１０、１１、１２及び１３において個々に説明される。図９は、構文スコープを現在のノードまたは後続のノードに基づいて適用することができて、宣言型プログラムモデルのブロックの内部から外部のスコーピングまたは上位から下位のスコーピングを特定するためにも適用することができることを示す。四分円が、図１０、１１、１２及び１３において更に詳細に示される。これによって、かかる構文スコーピングの選択に基づく得られる図表の表現を比較することができる。

【００３６】

図１０は、ドット及びアンパサンドが中心ブロックＡについて現れない、第１の構文レゾリューションの代替案１０００を示す。結果として、等価の有向グラフの図の表現は右側に表示される。

【００３７】

図１１は、ドットはないがアンパサンドが中心ブロックＡについて現れる第２の構文レゾリューションの代替案１１００を示す。結果として、等価の有向グラフの図の表現は右側に表示される。

【００３８】

図１２は、ドット及びアンパサンドの両方が中心ブロックＡに現れない第３の構文レゾリューションの代替案１２００を示す。結果として、等価の有向グラフの図の表現は右側に表示される。

【００３９】

図１３は、ドットはあるがアンパサンドが中心ブロックＡに現れない第３の構文レゾリューションの代替案１２００を示す。結果として、等価の有向グラフの図の表現は右側に表示される。上記のように、図表の表現は、有向グラフを宣言型プログラムコンストラクトのブロックの構文スコーピングを介して制御可能でかつ柔軟に形成することができる方法を説明する。

【００４０】
レゾリューションについてのいくつかの例示的で非限定的なルールに関して、
１．暗示的な上位レベルのスコープがある。
【００４１】
２．あらゆるブロックは、新規なスコープを導き、ラベルはスコープ内で定義される。
【００４２】
３．同じラベルがスコープ内で複数回定義される場合、全ての定義は一緒にされる。
【００４３】
４．絶対またはローカル
ａ．主要なドットを用いる参照は絶対であり、上位レベルのスコープ内の全ての一致しているラベルにバインドする。
ｂ．主要なドットのない参照はローカルであり、一致のある最内部のスコープ内の全ての一致しているラベルにバインドする。
【００４４】
５．LvalueまたはRvalue
ａ．主要なアンパサンドを有する参照は、参照が分解されるノードまたは複数のノードを返す。
ｂ．主要なアンパサンドのない参照は、参照が分解される全てのノードの全ての後続のものを返す。
【００４５】
アルゴリズムは、ラベルを介した同等関係を含む。これは一般にグラフに要求されない。しかしながら、テキスト形式において、参照は、全ての内部のノード（後続処理を有するノード）であるような識別子である。よって、文字列の同等を、比較のために用いることができるが、例えば、大文字と小文字を区別しない場合などの他の比較も可能である。
【００４６】
１つの非限定的な実施形態では、参照は、レゾリューションの間自身をトラバースしようとすることを可能にするが、これはエラーである点に注意される。従って、表現I{I}は、意味を持たない。
【００４７】
パスに関して、演算子のドット「.」は、スコープ内のラベルを調べ、その後続処理（または複数の後続処理）を返す。例えば、以下の記述は、Jack及びJillの例のうちのいずれかについて真である。
People.Jack.Age==23
People.Jill.Name==”Jill”
一般に、パスは、複数のノードに分解することができる。例えば、パスPeople.Jackは、ノードName、Age、Spouseのセットへ分解する。
【００４８】
Ｄグラフはまた、ファクタリングされた定義をサポートすることが有利である。その結果、構文的長針として、ノードの後続処理は全て一緒に定義される必要はない。例えば、以下の表現を検討する。

```
People {
      Jack {Jack {Name “Ｊａｃｋ”, Age 23},
      Jill {Name “Ｊｉｌｌ”, Age 23}
      }
```

等価のグラフの定義は以下である。
【００４９】

```
People {Jack {Name = “Ｊａｃｋ”, Age = 23}}
People {Jill {Name = “Ｊｉｌｌ”, Age = 25}}
```

更に以下のようにファクタリングされる。
【００５０】

People {Jack {Name = “Ｊａｃｋ”}},
People {Jack {Age = 23}},
People {Jill {Name = “Ｊｉｌｌ”}},
People {Jill {Age = 25}}

Ｄグラフのノードの簡単なセットの内容において、隣同士に書かれているので、このような方法で定義をファクタリングする能力は、特に役立つように見えないかもしれない。しかしながら、ファクタリングされた定義は、配列される必要はない。例えば、格納を分散することができる。しばしば、グラフの定義は非常に大きく、多くのファイルにわたって定義を広める能力は、定義をファクタリングすることで成し遂げることができる。

【００５１】

グラフ同士の間の同値関係は、以下のように定義される。テキスト表現では、ノードのあらゆるコンマ区切りのコレクションは、スコープを定義する。各スコープ内で、ラベルは複数回現れてもよい。テキストが表すグラフにおいて、あらゆるラベルの発現の全ての後続処理は、ラベルの後続処理である。例えば、以下の表現、

x {A{}, B{}, C{}} , x {D{}, E{}, F{}}

は、

x {A{}, B{}, C{}, D{}, E{}, F{}}

と同じグラフを表す。

【００５２】

これらのステップをトラッキングすると、以前の例は以下のように簡単になる。

People {Jack {Name = “Ｊａｃｋ”},
Jack {Age = 23},
Jill {Name = “Ｊｉｌｌ”},
Jill {Age = 25 }}

次に以下のようになる。

People {Jack {Name = “Ｊａｃｋ”, Age = 23},
Jill {Name = “Ｊｉｌｌ”, Age = 25}}

Ｄグラフを用いて、実施は、グラフが構築される時またはグラフがパス表現を用いてトラバースされる時のいずれかにいつでも、これらの定義を自由に構成することができる。ファクタリングは、格納をセグメント化する能力及び宣言をセグメント化する能力をシステムに与える。しかしながら、観察者は、全ての定義が一緒にされているかのように、グラフとインタラクションすることができる。

【００５３】

このことは、バイシミュレーションに類似している。バイシミュレーションは、システムに関係している、状態遷移システム同士の間の２項関係である。これは、１つのシステムがもう１つのシステムをシミュレーションするかまたはその逆にシミュレーションするという意味では、同様に機能する。２つのシステムは、これらがお互いの動きに一致する場合、バイシミュレーションであることは直感的である。この意味では、システムの各々は、観察者がもう１つと区別することができない。

【００５４】

階層化及びパスに関して、Ｄグラフの目的は、複数のパスまたはフェーズで生じ得る文字列からグラフを構築することである点に注意される。基本的なフローは、図１に示したように、文字列から木からグラフまたはその逆である。しかしながら、このフレームワーク内で、いつ参照を分解するべきかかつ同じラベルを用いて兄弟ノードの後続処理をいつ合体させるかに関する柔軟な範囲が存在する。上記の例示的な仕様は、レゾリューションに、いくつかの付加された複雑さが実施される必要がある合体の前に生じることを可能にするよう設計されている。しかしながら、このことは、グラフを格納する際により大きい柔軟性を提供し、従って実施するだけの価値がある。

【００５５】

別個だが関連する問題は、Ｄグラフとして見なされる既存のグラフにわたって起こる解

釈はどれくらいかということである。例えば、いずれかの任意のグラフは、Ｄグラフと見なすことができる。アクセスタイムまで合体するのを延期することによって任意のグラフが合体されたと見なされるのを可能にする。

【００５６】

上記のように、機能的な抽象化に関して、Ｄグラフは、ＸＭＬまたはＳ式のような基本的データ表現である。基本的データ表現において、表現の範囲内で機能的な抽象化を定義することは望ましい。しかしながら、これは、木だけに対応している、機械語（ＭＬ）またはＬＩＳＰのような標準機能のプログラミング言語に勝る。

【００５７】

明示的な参照に関して、書かれた参照は、任意でグラフが構築されるときに分解されるので、これらは別に処理することができて、異なる種類のエッジとして扱うことができる。例えば、このモデルにおいて、非参照エッジは木を形成し、参照は「２つのグラフ」モデルにおけるように木をトラバースすることができる別個のエッジのクラスに存在する。

【００５８】

様々な追加の例が、本明細書において記載されるＤグラフの様々な態様を強固にするために非限定的なやり方で以下に示される。上記のように、Ｄにおける基本的なデータ抽象化は、例えば、図１４の例示的な例１４００で示すように、ラベルを付けられた有向グラフである。

【００５９】

グラフは、ラベルを付けられたノード及び後続処理の順序のないコレクションから成る。ラベルは、例えば、Name、Jack、People及び「Jack」、23、2008-2-29などのリテラルから成る。この構造を有するグラフは、本明細書においてＤグラフと称される。従来のシステムと異なって、グラフは木に限定されておらず、図１５の例示的な有向グラフ１５００で示されるような円を含むことができる。

【００６０】

後続処理は、順序付けられていない。順序付けられたデータは、ノードのリストを用いて通常の方法で構築される。例えば、リストＡ、Ｂ、Ｃ、Ｄは以下のように表される。

【００６１】

【００６２】

後続処理のコレクションの範囲内で、ラベルがある場合は、エッジではなく、ラベルを付けられているノードで一意である。ラベルはまた、省略することができる。例えば、図１６の有向グラフ表現１６００を図１５の有向グラフ表現１５００と比較する。

【００６３】

グラフ構造は、任意であってもよい。表現及び型システムは、これらのグラフに構造を課すかまたは認識する。多くの他の可能な表現及び型システムがあるが、例示的で非限定的なやり方で、以下の説明は２つの表現及び１つの型システムを検討する。

【００６４】

本実施形態では、Ｄグラフのテキスト形式の表現は、以下の内容自由の文法に従う。

```
DGraphList:
    DGraph
    DGraphList , DGraph
DGraph:
    Literal
    Qualifiedldentifier
    Label = DGraph
    Labelopt { DGraph }
```

```
Label:
    Identifer TypeAscription_opt
    Qualifiedldentifier：
      Identifer
        Qualifiedldentifier . Identifier
```

例えば、図１４のグラフ１４００は、以下のようにテキスト形式で表すことができる。

【００６５】

```
People {
    Jack {Name {“Ｊａｃｋ”}, Age {23}}
    Jill {Name {“Ｊｉｌｌ”}, Age {25}}
}
```

同じ標準的なＤグラフについての多くの可能なテキスト形式の表現が存在する。構文のショートカットとして、等号「=」は、ラベルがただ１つの後続処理を有するときに用いることができる。以下のテキストは、上記のテキスト及びグラフ１４００として正確に同じＤグラフを表す。

```
People {
   Jack {Name = “Ｊａｃｋ”, Age = 23},
   Jill {Name = “Ｊｉｌｌ”, Age = 25}
}
```

上記のように、ラベルは、以下のようにコンパクトな表現のために省略することができる。

```
People {
   {Name = “Ｊａｃｋ”, Age = 23},
   {Name = “Ｊｉｌｌ”, Age = 25}
}
```

ラベルは省略することができるが、これらはなお役立つ。ラベルなしで、木構造グラフの限定された系統だけを構築することは可能である。

【００６６】

上記のように、Ｄグラフは、ファクタリングされた定義をサポートする。即ち、構文の長針のように、ノードの後続処理が全て一緒に定義される必要はなくて、後続処理の各コレクションはスコープを定義する。MemberAccess演算子のドット「.」は、スコープ内のラベルを調べて、その後続処理（または複数の後続処理）を返す。

【００６７】

スコープメカニズムの別の使用法は、図１５の例１５００におけるように、グラフの他の部分への参照を構築することである。以下のテキストは、そのグラフを表す。

```
People {
   Jack {Name = “Ｊａｃｋ”, Age = 23, Spouse = Jill},
   Jill {Name = “Ｊｉｌｌ”, Age = 25, Spouse = Jack}
}
```

（後続処理のない）アトミックな値として用いられると、識別子はグラフ内の別のノードへの参照である。参照は、最内部から最外部まで進む通常の構文スコーピングのレゾリューションルールを用いて分解される。

【００６８】

更に、以下の例は、上記及び例１５００と同じグラフを表わす。

```
People {
   Jack {Name = “Ｊａｃｋ”, Age = 23, Spouse = People.Jill},
   Jill {Name = “Ｊｉｌｌ”, Age = 25, Spouse = People.Jack}
}
```

Jackの定義の範囲内で、一致しているラベルは存在しないので、分解手順は最初に、Jackの後続処理の範囲内で識別子Jillを探す。次に手順は、Jackの先行オペレーションPeop

leの後続処理の範囲内で調べる。この場合、ラベルの一致が存在し、それはJack.Spouse及びPeopleの両方の後続処理になる。

【００６９】

非限定的な技術的詳細として、マスキングのいくつかのケースを処理するために、メンバーアクセス演算子は、左辺のオペランド（例えば、プレフィックス）なしに用いられて検索を最外部のスコープから開始できることが注目される。

【００７０】

例えば、以下は、上記の２つのグラフ及び例１５００と同じグラフを表す。

```
People {
   Jack {Name = “Ｊａｃｋ”, Age = 23, Spouse = .People.Jill},
   Jill {Name = “Ｊｉｌｌ”, Age = 25, Spouse = .People.Jack}
}
```

型システムは、グラフ内の構造を認識して（または課して）解釈を容易にする。多くの他の型システムがあるが、このセクションは、Ｄグラフについての１つの可能な型システムを定義する。よって、詳細は、包括的概念に対して非限定であるとみなされなければならない。リテラルであるSimpleTypeについての簡単な型システムが与えられると、型の宣言のための以下の文法が付随する。

```
Type:
    SimpleType
    HomogeneousCollection
    Entity
    Qualifiedldentifier
```

同種コレクションについて、同種コレクションの後続処理は、同じ型Ｔの中になければならない。以下は、同種コレクションの例である。

```
{1, 2, 3}
{“A”, “B”, “C”}
People {
   Jack {Name = “Ｊａｃｋ”, Age = 23},
   Jill {Name = “Ｊｉｌｌ”, Age = 25}
}
```

コレクションの基数は、演算子？、+、*または#m..nを用いて規定される。３から７までのT（包括的）の同種コレクションは、T#3..7と書かれる。任意の数のSの同種コレクションは、以下の文法に基づいてS*と書かれる。

```
HomogeneousCollection：
    Type # LowerBound .. UpperBound
    Type # .. UpperBound
    Type # LowerBound ..
    Type *
    Type +
    Type ?
```

..の左側の整数を省略することによって、下限が零であるよう定義される。右側の整数を省略することによって、上限が無限であるよう定義される。星印「*」は、境界を零から無限であるよう定義する。プラス印「+」は、境界を１から無限であるよう定義する。疑問符印「?」は、境界を零から１であるよう定義する。

【００７１】

エンティティは、必要なラベルのセット及びこれらの後続処理の型を特定する。後続処理の型は、各ラベルについて異なっていてもよい。以下は、エンティティの例である。

```
Jack {Name = “Ｊａｃｋ”, Age = 23}
StarWarsI {Title = “Star Wars: A New Hope”, Duration = 121}
```

エンティティは、オプションの型帰属（ｔｙｐｅ　ａｓｃｒｉｐｔｉｏｎ）及びオプシ

ョンのデフォルト値を用いてセミコロンで終わるラベルのリストである。宣言の順序は無関係だが、全てのラベルははっきりと異なる。エンティティは、以下のように特定される。

```
Entity:
    {ElementList}
ElementList：
    Element
    ElementList Element
Element：
    Identifier TypeAscriptionopt;
    Identifier TypeAscriptionopt DefaultValueopt
TypeAscription：
    :Type
DefaultValue：
    = DGraph ;
     { DGraphList }
```

その後続処理がそれぞれ型テキスト及び数である２つの後続処理Name及びAgeを有するエンティティは、

```
{Name：Text; Age: Number;}
```

と書かれる。

【００７２】

型帰属は、ノードの後続処理に所定の型に忠実であるよう強いる。例えば、以下は有効な型帰属である。

```
People : {Name：Text; Age: Number;} * {
  Jack {Name {“ Ｊａｃｋ ”}, Age {23}},
  Jill {Name {“ Ｊｉｌｌ ”}, Age {25}}
}
```

対照的に、以下は無効な型帰属である。

```
People : {City：Text; Zip: Number;} * {
  Jack {Name {“ Ｊａｃｋ ”}, Age {23}},
  Jill {Name {“ Ｊｉｌｌ ”}, Age {25}}
}
```

あらゆる定義のように、ノードについての型帰属は、例えば以下のようにファクタリングすることができる。

```
People : {City：Text; Zip: Number;} * ,
People {
  Jack {Name {“ Ｊａｃｋ ”}, Age {23}},
  Jill {Name {“ Ｊｉｌｌ ”}, Age {25}}
}
```

Ｄでは、型は値のコレクションとして定義される。このコレクションは、HomogeneousCollection及びEntityから構成される型表現を用いて暗示的に定義することができる。あるいは、グラフのいかなるノードの後続処理もコレクションを定義し、これも型として用いることができる。以下を検討する。

```
People : {City：Text; Zip: Number; Spouse : People;} *
```

このことは、Spouseのノードの後続処理がPeopleのノードの後続処理のうちの１つであることを必要とする。従って、以下は有効である。

```
People : {City：Text; Zip: Number; Spouse : People;} *
People {
   Jack {Name = “ Ｊａｃｋ ”, Age = 23, Spouse = Jill},
   Jill {Name = “ Ｊｉｌｌ ”, Age = 25, Spouse = Jack}
```

}
People.Jack.Spouseの後続処理はJillである。JillはPeopleの後続処理であるので、これは型を満たす。
【００７３】
更に、型システムは、グラフを介して解釈の追加のレベルを課することができる。コンパイラは、この解釈を用いて、型コンストレイントを満足させるために必要な暗示的なエッジを追加する。次の例を考える。

```
People : {City：Text; Zip: Number; Spouse : People;} *
People {
   Jack {Name = “Ｊａｃｋ”,
      Age = 23,
      Spouse = {Name = “Ｊｉｌｌ”, Age = 25, Spouse = Jack}
}
   }
```

この例は、上記の例に類似している。後続処理Jackを有するノードPeopleが存在する。この宣言についてのグラフを描くことによって、図１７の例１７００が生成される。レイアウトは異なっているが、グラフ１７００はグラフ１５００とほとんど同形である。それは、同じノード数を有する。それは１つのラベルであるJill及びPeopleから以前にJillとラベルを付けられていたノードへの１つのエッジが欠けている。
【００７４】
ところが実際は、グラフ１７００は、Peopleノードで型帰属に従っていない。欠けているラベルは問題でない。エンティティのラベルは必要でない。唯一の問題は、欠けているエッジである。この１つの特別なケースにおいて、コンパイラはエッジを追加することができる。型帰属は、Spouseの後続処理もPeopleの後続処理であることを必要とするからである。
【００７５】
図１８のグラフ１８００で示すように、この特別のケースは、テキスト形式の表現の入れ子にされたグラフを宣言するのを容易にして、表現のショートカットを可能にする。追加されたエッジを用いて、２つのグラフは、エンティティの値の任意のラベルをつける同形のモジュロである。
【００７６】
（例示的な宣言型プログラミング言語）
誤解を避けるために、Ｄプログラミング言語などの宣言型プログラミング言語に関してこのサブセクションにおいて提供される付加的な内容は、非網羅的であって非限定的であるとみなされることになっている。以下に記載される疑似コードのセットについての特別な例の断片は、実例及び説明のためだけであって、様々な詳細において上記の宣言型プログラミングモデルのための有向グラフ構造の実施形態を限定するとみなされることになっていない。
【００７７】
図１９において、例えばＤプログラミング言語に基づくモデルなどの、宣言型モデルについての例示的な一連のプロセスが提供される。示すように、一連のプロセス１９００は、コンパイラ１９２０、パッケージングコンポーネント１９３０、同期コンポーネント１９４０及び複数のレポジトリ１９５０、１９５２、...、１９５４の結合を含むことができる。かかる実施形態内で、コンパイラ１９２０に入力されるソースコード１９１０は、Ｄプログラミング言語などの宣言型プログラミング言語で書かれた宣言型実行モデルを表す。Ｄプログラミング言語については、例えば、ソースコード１９１０で具現化される実行モデルがコンストレイントベースの分類または構造上の分類に続くことが利点であり、かつ／またはコードの開発を簡単にするために順序無関係即ち順序のない実行モデルを具現化することが有利である。
【００７８】

コンパイラ１９２０は、ソースコード１９１０を処理して、各ソースコードについての後処理された定義を作成することができる。他のシステムは命令型のフォーマットまでコンパイルを行なうが、変換される間、ソースコードの宣言型のフォーマットは保存される。パッケージングコンポーネント１９３０は、Ｄプログラミング言語の場合におけるＤ＿Ｉｍａｇｅファイルなどの画像ファイルとして、後処理された定義をパッケージングする。画像ファイルは特定のレポジトリ１９５０、１９５２、．．．、１９５４にインストールすることができる。画像ファイルは、必要なメタデータの定義及びこれらの宣言型ソースモデルと共に複数の変換された中間生成物を格納するための拡張可能なストレージの定義を含む。例えば、パッケージングコンポーネント１９３０は、特定のメタデータの特性を設定することができて、画像ファイル内のコンテンツ部分としてコンパイラの出力した中間生成物と共に宣言型ソース定義を格納することができる。

【００７９】

Ｄプログラミング言語について、パッケージングコンポーネント１９３０が用いるパッケージングフォーマットは、ＥＣＭＡのオープンパッケージングコンベンション（ＯＰＣ）標準に準拠している。当業者は、この標準が本質的に圧縮、グループ化、署名などの特徴を提供すると直ちに認めることができる。この標準はまた、画像ファイルを標準のプログラムツールを介して操作できるようにする一般的なプログラムモデル（ＡＰＩ）を定義する。例えば、.NET Frameworkにおいて、ＡＰＩは、「System.IO.Packaging」の名前空間内で定義される。

【００８０】

同期コンポーネント１９４０は、画像ファイルを管理するために用いることができるツールである。例えば、同期コンポーネント１９４０は、入力として画像ファイルをとって、これを参照画像ファイルのセットにリンクさせることができる。その中間にまたはその後に、パッケージングされた中間生成物を引き出し、これらを処理し、同じ画像ファイル内により多くの中間生成物を加えることで、画像ファイルを介して動作する（書き換えツール、最適化ツール等のような）いくつかのサポーティングツールが存在してもよい。これらのツールはまた、画像ファイルについてのいくつかのメタデータを操作して、画像ファイルの状態を変更することができる。例えば、その完全性及びセキュリティを確実にするために画像ファイルにデジタル署名することなどである。

【００８１】

次に、配置ユーティリティが画像ファイルを配置し、インストールツールがレポジトリ１９５０、１９５２、．．．、１９５４内で実行中の実行環境に画像ファイルをインストールする。一旦画像ファイルが配置されると、様々な配置後のタスクに従うことができる。様々な配置後のタスクは、エクスポート、ディスカバリ、サービス、バージョニング、アンインストール及びより多くのものを含む。Ｄプログラミング言語については、セキュリティ、拡張性、スケーラビリティ及び性能のような企業レベルの産業上の要件をなお満たしながら、パッケージングフォーマットは全てのこれらのオペレーションのためのサポートを提供する。１つの実施形態において、レポジトリ１９５０は、リレーショナルデータベース管理システム（ＲＤＢＭＳ）のコレクションであってもよいが、いずれかのストレージに含まれてもよい。

【００８２】

１つの実施形態において、本明細書において説明されている方法は、コンストレイントベースの型システムを有するプログラミング言語を用いて動作可能である。かかるコンストレイントベースのシステムは、単なる従来のノミナルな型システムで可能なのではない機能を提供する。図１３～１４において、ノミナルな型の実行システムは、本発明の実施形態に基づいてコンストレイントベースの型の実行システムと比較される。示したように、ノミナルなシステム１３００は、特定の型をあらゆる値に割り当て、一方、コンストレイントベースのシステム１３１０の値は、無数の型のうちのいずれかと一致することができる。

【００８３】

Ｄプログラミング言語などの、ノミナルな型の実行モデルと本明細書において説明された宣言型プログラミング言語に基づくコンストレイントベースの型のモデルとの対照を示すために、各モデルの型宣言についての例示的なコードが以下で比較される。

【００８４】

最初に、ノミナルな型の実行モデルに関して、以下の例示的なＣ＃コードが示される。

```
Class A
{
    public string Bar;
     public int Foo;
}
Class B
{
    public string Bar;
    public int Foo;
}
```

この宣言について、柔軟性のない型-値関係は、Ｂａｒ及びＦｏｏのこれらのフィールドの値が同一であるとしてもＡの値及びＢの値が比較にならないとみなされる関係において存在する。

【００８５】

対称的に、コンストレイントベースのモデルに関して、以下の例示的なＤコード（更に詳細に以下で検討される）は、オブジェクトが多くの型に一致できる方法を示す。

```
type A { Bar : Text; Foo : Integer; }
type B { Bar : Text; Foo : Integer; }
```

この宣言について、型-値関係は、型Ａに一致する全ての値が同様にＢに一致しまたこの逆も一致するように、はるかに柔軟である。更に、コンストレイントベースのモデルにおける型は、互いの上に重ねることができ、これは、例えば、様々なＲＤＢＭＳにわたってプログラミングするのに役立つ柔軟性を提供する。実際に、コンストレイントベースのモデルにおける型は最初に世界中の全ての値を含むので、特定の値は、値が型の宣言において体系化されたコンストレイントに違反していない全ての型と一致する。よって、宣言type T : Text where value<128で定義される型と一致する値のセットは、「整数」のコンストレイントまたは「値＜１２８」のコンストレイントに違反しない「世界の全ての値」を含む。

【００８６】

よって、１つの実施形態において、ソースコードのプログラミング言語は、Ｄプログラミング言語で実施されるような、上記のコンストレイントベースの型システムを含む純粋な宣言型言語である。

【００８７】

別の実施形態において、本明細書において説明される方法はまた、順序無関係即ち順序のない実行モデルを有するプログラミング言語を用いて動作可能である。上記のコンストレイントベースの実行モデルと同様に、かかる順序無関係の実行モデルは、例えば、様々なＲＤＢＭＳを介してプログラミングするのに役立ち得る柔軟性を提供する。

【００８８】

図２２～２３において、説明のために、順序付けられた実行モデルに基づくデータ・ストレージ・アブストラクションは、順序無関係の実行モデルに基づくデータ・ストレージ・アブストラクションと比較される。例えば、図２２のデータ・ストレージ・アブストラクション２２００は、順序付けられた実行モデルに基づいて作成されるリストＦｏｏを表し、一方、図２３のデータアブストラクション２２１０は、順序無関係の実行モデルによって作成される類似のリストＦｏｏを表す。

【００８９】

説明したように、データ・ストレージ・アブストラクション２２００及び２２１０の各

々は、３つのＢａｒ値（即ち、「１」、「２」及び「３」）から成るセットを含む。しかしながら、データ・ストレージ・アブストラクション２２００は、これらのＢａｒ値が特定の順番で入力される／リストに挙げられることを必要とし、一方、データ・ストレージ・アブストラクション２２１０は、かかる要求を行わない。代わりに、データ・ストレージ・アブストラクション２２１０は、単にＩＤを各Ｂａｒ値に割り当てる。これらのＢａｒ値を入力し／リストに挙げる順序は、目標とされるレポジトリに対して観察できない。例えば、データ・ストレージ・アブストラクション２２１０は、よって以下の順序無関係のコードから得られた。

f： Foo* ={Bar = “１”}；
f： Foo* ={Bar = “２”}；
f： Foo* ={Bar = “３”}；

しかしながら、データ・ストレージ・アブストラクション２２１０は、以下のコードから得ることができる。

f： Foo* ={Bar = “３”}；
f： Foo* ={Bar = “１”}；
f： Foo* ={Bar = “２”}；

更に、上記の２つのコードの各々は、以下のコードに機能的に等価である。

f： Foo* ={ {Bar = “２”}，{Bar = “３”}，{Bar = “１”}}；

上記のコンストレイントベースの型で順序のない実行モデルと互換性を持つ例示的な宣言型言語は、便宜上「Ｄ」と本明細書で称される場合があるＤプログラミング言語である。Ｄプログラミング言語は本発明の出願人によって開発された。しかしながら、Ｄに加えて、他の類似の宣言型プログラミング言語を用いることができかつ本発明のユーティリティは、上記の有向グラフ構造の実施形態のうちのいずれか１つまたは複数があてはまるいずれかの単一のプログラミング言語に限定されないと理解されることになっている。この点について、Ｄに関するいくつかの付加的な内容が以下で提供される。

【００９０】

上記のように、Ｄは、データと協働する宣言型言語である。Ｄは、ユーザに、書き込み可能でかつ読み出しが可能である従来のテキスト構造を用いて自分たちのデータを構築してクエリしたい方法を決定させる。１つの非限定的な態様において、Ｄプログラムは、正式にはコンパイルユニットとして知られる１つまたは複数のソースファイルを含む。ソースファイルは、ユニコード文字の整然とした順序である。ソースファイルは通常、ファイルシステムにおいてファイルとの１対１の対応があるが、この対応は必要でない。最大ポータビリティについて、ファイルシステム内のファイルがＵＴＦ－８符号化を用いてコード化されることが薦められる。

【００９１】

概念的に言えば、Ｄプログラムは、４つのステップを用いてコンパイルされる。１）字句解析。ユニコード入力されたキャラクタの文字列をトークンの文字列に翻訳する（字句解析が前処理指令を評価して実行する）。２）統語解析。トークンの文字列を抽象構文木に翻訳する。３）意味解析。抽象構文木における全てのシンボルを分解し、型は構造をチェックして意味グラフを作成する。４）コード生成。これは、いくつかの目標ランタイム（例えば、画像を作るＳＱＬ）についての意味グラフから実行可能命令を生成する。更なるツールは、イメージを連結して、これらをランタイムにロードすることができる。

【００９２】

宣言型言語として、Ｄはデータが格納されるかまたはアクセスされる方法を義務付けないし、（ＸＡＭＬなどの領域特定言語と対照的に）特定の実施技術も義務付けない。むしろ、ユーザに、自分たちの所望が所定の技術またはプラットフォームに一致する方法を特定する必要なしに、自分たちのデータから自分たちが所望するものを書き留めるのを可能にするようＤは設計されていた。Ｄは、Ｄのコンストラクトが所定の環境で表されて実行される方法を制御するための豊富な宣言型サポートまたは必須のサポートを実施例が提供するのを決して禁止せず、よって、豊富な開発の柔軟性を可能にすると述べられている。

【００９３】
Ｄは、３つの基本的な概念を基にしている。即ち、値、型及び範囲である。これらの３つの概念は、以下のように定義することができる。１）値はＤ言語の規則に従うデータである。２）型は値のセットを記述する、３）範囲は値のための動的格納を提供する。
【００９４】
一般に、Ｄは、データの型をデータの格納／範囲から切り離す。所定の型は、複数の範囲からのデータを記述するためにかつ計算の結果を記述するために用いることができる。これは、ユーザに、最初に型を書き留めるのを開始させて、その後どこで対応する値を置くかまたは計算するかを決定できるようにする。
【００９５】
どこに値を置くべきかについて決定する主題に関して、Ｄ言語は、実施例が宣言された範囲をＲＤＢＭＳなどの外部のストアにマッピングする方法を特定しない。しかしながら、Ｄは、かかる実施例を可能にするよう設計されていて、リレーショナルモデルと互換性を持っている。
【００９６】
データ管理に関して、Ｄは、範囲のコンテンツを変更するためのコンストラクトを有していない機能言語である。しかしながら、Ｄは、範囲のコンテンツが外部の（Ｄへの）刺激を介してかつ任意で変わることができると予測し、Ｄは更新データについての宣言型構造概念を提供するよう修正することができる。
【００９７】
検証または割り当てのために値を分類する方法を書き留めることは多くの場合望ましい。Ｄにおいて、値は型を用いて分類される。Ｄの型は、受け入れ可能な値または整合する値のコレクションを記述する。更に、Ｄの型は、どの値が特定の内容（例えば、オペランド、格納先）に現れることができるかを抑制するために用いられる。
【００９８】
Ｄによって、型がコレクションとして用いられるようになる。例えば、「in」演算子は、
l in Number
“Hello, world” in Text
などの、値が所定の型に位置するかどうかを検査するために用いることができる。
【００９９】
ビルトイン型の名前は、Ｄ言語において直接利用可能である点に注意しなければならない。しかしながら、型についての新しい名前はまた、型宣言を用いて導かれてもよい。例えば、下記の型宣言は、「Text」の簡単な型についての同義語として型名「My Text」を導入する。
type [My Text] : Text;
ここで利用可能なこの型名を用いて、以下のコードを書くことができる。
“Hello, world” in [My Text]
既存の型についてカスタム名を導入することは役立つが、述部を、
type SmallText : Text where value.Count<7
などの基本的な型に適用することは更に役立つ。
【０１００】
この例では、可能な「Text」値の世界は、それらに強制された、７文字未満の値が含まれる値に抑制される。従って、以下の記載は、この型の定義について真を保持する。
“Terse” in SmallText
!(“Vervose” in SmallText)
型宣言は以下を構成する。
type TinyText : SmallText where value.Count<6;
しかしながら、この例では、この宣言と等価である。
Type TinyText： Text where value.Count<6;

型の名前が存在し、Ｄ宣言または表現がそれを参照できることが注目される。名前のいくつでも、同じ型に割り当てることができ（例えば、Text where value.Count<7）、所定の値は、これらの全てに一致するかまたはこれらのいずれにも一致しない。例えば、この例を検討する。

```
type A : Number where value < 100;
type B : Number where value < 100;
```

これらの２つの型の定義が与えられると、以下の表現の両方、

```
l in A
l in B
```

は、結果が真である。以下の第３の型

```
type C : Number where value> 0;
```

が導入される場合、
以下を記述することができる。

```
l in C
```

Ｄの一般原理は、所定の値がいかなる数の型にも一致することができるということである。これは、多くのオブジェクトベースのシステムが機能する方法からの脱却であり、値は初期化時間に特定の型にバインドされ、型が定義されるときに特定されたサブタイプの有限集合のうちの一部分である。

【０１０１】

検討を要する別の型に関連したオペレーションは、型帰属の演算子（：）である。型帰属の演算子は、所定の値が特定の型に一致することを示す。

【０１０２】

一般に、表現内に値が見られる場合に、Ｄは、適用されている演算子／関数についての宣言された結果の型に基づくその値の予想される型についてのいくつかの概念を有する。例えば、論理積演算子（&&）の結果は、型「Logical」と一致することになっていると宣言される。

【０１０３】

通常、異なっている要件を有する別の内容においてその値を用いることである、追加のコンストレイントを所定の値に適用することは時々役立つ（または必要でさえある）。例えば、以下の型の定義を検討する。

```
type SuperPositive : Number where value > 5;
```

オペランドとして型「SuperPositive」の値を受け入れると宣言されている「CalcIt」という関数が存在すると仮定するならば、Ｄにおいてこのような表現を可能にすることが望ましい。

```
D:
      CalcIt(20)
      Calclt(42 + 99)
```

及び、このような表現を禁止する。

【０１０４】

```
      CalcIt(-l)
      CalcIt(4)
```

事実、Ｄはまさにこれらの４つの例に欠けることを行う。これは、これらの表現が定数を超えて組み込まれた演算子に関してこれらのオペランドを表すからである。表現についてのＤソーステキストがわずかな犠牲で見つけられた時に、表現の正当性を決定するために必要な情報の全てを直ちに利用できる。

【０１０５】

しかしながら、表現が、データ及び／またはユーザ定義の関数の動的ソースを利用する場合、型帰属の演算子は、値が所定の型に一致できることを示すために用いられる。

【０１０６】

型帰属の演算子が値と協働する方法を理解するために、型「Text」のオペランドを受け

入れてオペランドの母音の数を示す型「Number」の値を返すと宣言される第２の関数、「GetVowelCount」が想定される。

【０１０７】

その結果が５より大きいか否かは、「GetVowelCount」の宣言に基づいて分からないので、以下の表現は適法なＤの表現でない。

CalcIt(GetVowelCount(someTextVariable))

「GetVowelCount」の宣言された結果の型（数）が「CalcIt」SuperPositive）の宣言されたオペランド型に一致しない値を含むので、表現は適法でない。この表現は、誤って書かれたと推測することができる。

【０１０８】

しかしながら、この表現は、型帰属の演算子を用いて以下の（適法な）表現に書き直すことができる。

Calclt((GetVowelCount(someTextVariable)：SuperPositive))

この表現によって、Ｄは、型「SuperPositive」に一致する値を得ることができることを知っている「GetVowelCount」関数について十分理解されていると知らされる。要するに、プログラマは、Ｄがしていることを彼／彼女が知っているとＤに伝えている。

【０１０９】

しかしながら、「GetVowelCount」関数が機能する方法をプログラマが誤った場合をプログラマが知らない場合、特定の評価は負数をもたらすかもしれない。「CalcIt」関数は「SuperPositive」に一致する値を受け入れるだけであると宣言されたので、システムは、渡された全ての値が５より大きいことを保証する。このコンストレイントが決して破られないことを保証するために、システムは、評価される場合に失敗する可能性を有する動的コンストレイントテストを挿入することができる。この失敗は、Ｄのソーステキストが（CalcIt(-l)を用いるケースと同様に）最初に処理されると起こらない。むしろ、それは、表現が実際に評価される場合に起こる。

【０１１０】

この点について、Ｄの実施例は通常、Ｄ文書の第１の表現が評価される前にあらゆるコンストレイント違反を報告しようとする。これは静的施行と称されて、実施例は、構文エラーによく似ているこれを明らかにすることができる。しかしながら、いくつかのコンストレイントは、生のデータに対して実施することができ、従って、動的施行を必要とすることができるだけである。

【０１１１】

この点について、Ｄは、ユーザが自分たちの意図を書き留めて「うまくいくように」Ｄの実施例への負担となるのを簡単にする。オプションとして、特定のＤ文書がさまざまな環境において用いられるようにするために、十分に特徴付けられたＤ実施例が、動的コンストレイント違反のコンストレイント及び動作コストを減じるために正確さについての動的施行に依存するＤ文書を拒否するよう設定できる。

【０１１２】

Ｄに関する更なる背景について、型コンストラクタは、コレクション型を特定するために定義することができる。コレクション型のコンストラクタは、コレクションが含むことができる要素の型及びカウントを制限する。全てのコレクション型は、固有の型「Collection」を介した制限である。例えば、全てのコレクションの値は、以下の表現に一致する。

{} in Collection

{1, false} in Collection

! (“Hello” in Collection)

最後の例は、コレクション型が単純型と重ならないことを示す。コレクション型及び単純型の両方に一致する値は存在しない。

【０１１３】

コレクション型のコンストラクタは、要素の型及び受け入れ可能な要素のカウントの両

方を特定する。要素のカウントは通常、３つの演算子のうちの１つを用いて特定される。
T* - ０以上のT
T+ - １以上のT
T#m..n - mとnとの間のT
コレクション型コンストラクタは、Kleene演算子を用いることができるかまたは固有の型Collectionを介したコンストレイントとしての普通の書き方で書くことができる。即ち、以下の型の宣言は、同じコレクション値のセットを記述する。

```
type SomeNumbers : Number+;
type TwoToFourNumbers : Number#2..4;
type ThreeNumbers : Number#3;
type FourOrMoreNumbers : Number#4..;
```

これらの型は、これらの普通の書き方の定義として同じ値のセットを記述する。

```
type SomeNumbers : Collection whrere value.Count>=1
                   && item in Number;
type TwoToFourNumbers : Collection wherer value.Count >= 2
                 && value.Count <= 4
                 && item in Number,
type ThreeNumbers : Collection where value.Count == 3
                 && item in Numbe,
type FourOrMoreNumbers : Collection where value.Count >= 4
                 && item in Numbe,
```

どの形式が型を宣言するために用いられるかに関係なく、以下の表現を記述することができる。

```
! ( {} in TwoToFourNumbers)
!({“One”, “Two”, “Three”} in TwoToFourNumbers)
{1、2、3} in TwoToFourNumbers
{1, 2, 3} in ThreeNumbers
{1、2、3、4、5} in FourOrMoreNumbers
```

コレクションの型のコンストラクタは、「where」演算子を用いて構成し、以下の型チェックが、

```
{1、2} in (Number where value <3)* where value.Count % 2==0
```

の後に続くことを認める。内側の「where」演算子がコレクションの要素にあてはまり、外側の演算子がコレクション自体に当てはまることが注目される。

【０１１４】

コレクションの型の通りに、コンストラクタは、どんな種類のコレクションが所定の内容において有効か特定するために用いることができ、同じことを、エンティティ型を用いてエンティティについて行うことができる。

【０１１５】

この点について、エンティティ型は、エンティティの値のセットについて予想されるメンバーを宣言する。エンティティ型のメンバーは、フィールドとしてまたは計算値としてのいずれかで宣言することができる。フィールドの値は、格納される。計算値の値が計算される。エンティティ型はエンティティ型を介して制限され、エンティティ型はＤ標準ライブラリにおいて定義される。

【０１１６】

以下は、簡単なエンティティ型である。

```
type MyEntity : Language.Entity;
```

型「MyEntity」は、いずれのフィールドも宣言しない。Ｄにおいて、エンティティ型は、型に一致するエンティティの値が名前を型内で宣言していないフィールドを含むことができるという点で、オープンになっている。よって、以下の型テストは、

```
{X = 100, Y = 200} in MyEntity
```

は、「MyEntity」型が、Ｘ及びＹと称されるフィールドについて何も記述しないときに、結果は真である。

【０１１７】

エンティティ型は、１つまたは複数のフィールド宣言を含むことができる。少なくとも、フィールド宣言は、例えば、

type Point {X; Y;}

などの予想されるフィールドの名前を記述する。

【０１１８】

この型定義は、これらのフィールドの値に関係なく少なくともＸ及びＹと称されるフィールドを含むエンティティのセットを説明する。これは、以下の型テストが結果が真である事を意味する。

{X = 100, Y = 200} in Point

{X = 100, Y = 200, Z = 300} in Point // more fields than expected OK

!({X = 100} , in Point) // not enough fields - not OK

{X = true, Y = “Hello, world”} in Point

最後の例は、「Point」型がＸフィールド及びＹフィールドの値をコンストレイントしないことを示す。即ち、あらゆる値が可能である。数の値までＸ及びＹの値をコンストレイントする新しい型は、以下のように示される。

type NumericPoint {

X ： Number;

Y ： Number where value> 0

}

型属性の構文は、Ｘフィールド及びＹフィールドの値が型「Number」に一致すべきであると示すために用いることができることに注目される。これを所定の位置に用いると、表現は結果として真である。

{X = 100, Y = 200} in NumericPoint

{X = 100, Y = 200, Z = 300} in NumericPoint

!({X = true, Y = “Hello, world”} in NumericPoint)

!({X = 0, Y = 0} in NumericPoint)

簡単な型の検討において見られたように、型の名前が存在し、Ｄ宣言及び表現がその名前に関連することができる。これは、NumericPoint及びPointの定義がたとえ無関係であっても、以下の型テストの両方が後に続くからである。

{X = 100, Y = 200} in NumericPoint

{X = 100, Y = 200} in Point

Ｄのフィールドは、値を保持する名前を付けられた格納ユニットである。Ｄは、開発者がエンティティのイニシャライザの一部としてフィールドの値を初期化できるようにする。しかしながら、一旦初期化されると、Ｄは、フィールドの値を変更するいずれかのメカニズムを特定しない。Ｄにおいて、フィールド値へのあらゆる変更がＤのスコープの外側で起こると推測される。

【０１１９】

フィールド宣言は、フィールドについてのデフォルトの値が存在することを示すことができる。デフォルトの値を有するフィールド宣言は、特定された対応するフィールドを有する整合エンティティを必要としない（上記のフィールド宣言は、オプションフィールドということもある）。例えば、以下の型の定義に関して、

type Point3d {

X ： Number ;

Y ： Number ;

Z = -1 ： Number; // default value of negative one

}

Zフィールドは、デフォルトの値を有しているので、以下の型テストが後に続く。

{X = 100(Y = 200)}in Point3d
更に、型帰属の演算子が以下のように値に適用される場合、
（{X = 100, Y = 200} ： Point3d）、
Ｚフィールドを、以下のようにアクセスすることができる。
（{X = 100, Y = 200} ：Point3d）.Z
この場合、この表現は値-1を生じ得る。
【０１２０】
別の非限定的な態様において、フィールド宣言が対応するデフォルト値を有しない場合、整合エンティティは、そのフィールドについて値を特定しなければならない。デフォルト値は通常、「Point3d.」のＺフィールドについて示される明示的な構文を用いて書き留められる。フィールドの型は、ヌルであってもよいかまたは零から多へのコレクションのいずれかである場合、オプションのためのヌル及びコレクションのための {} の宣言フィールドについての暗示的なデフォルトの値が存在する。
【０１２１】
例えば、以下の型を検討する。

```
type PointND（
X ： Number;
Y ： Number;
Z ： Number;?;  // Z is optional
BeyondZ : Number*; //BeyondZ is optional too
}
```

次に、再び、以下の型テストが続く。
{X = 100, Y = 200} in PointND
及び「PointND」 を値に帰属させることによって、これらのデフォルトが生じる。
({X = 100, Y = 200} : PointND）.Z == null
({X = 100, Y = 200} : PointND）.BeyondZ == {}
零から１までのコレクションまたはヌルが可能な型対明示的なデフォルトの値をモデルのオプションフィールドに対して用いることについての選択は、通常、スタイルのうちの１つに及ぶ。
【０１２２】
計算値は名付けられた表現であり、その値は、格納されるよりはむしろ算出される。かかる計算値を宣言する型の例は、以下の通りである。

```
type PointPlus {
 X ： Number;
 Y ： Number;
// calculated value
IsHigh0 : Logical {Y > 0;}
}
```

セミコロンで終わるフィールド宣言と異なり、計算値の宣言は、最後に中括弧で囲まれた表現が付いている。
【０１２３】
フィールド宣言のように、計算値の宣言は、この例のように、型帰属を省略することができる。

```
type PointPlus {
X ： Number;
Y ： Number;
// a calculated value with no type ascription
InMagicQuadrant() {IsHigh && X > 0;}
IsHigh0 ： Logical  {Y > 0;}
}
```

別の非限定的な態様において、型が計算値に明示的に帰属されていない場合、Ｄは、基本的な表現の宣言された得られた型に基づいて、自動的に型を推測することができる。この例では、表現で用いられる論理及び演算子が「Logical」に戻ると宣言されたので、「InMagicQuadrant」計算値はまた、「Logical」値を生じるとみなされる。

【０１２４】

上記で定義されて用いられる２つの計算値は、エンティティ値自体以外のそれらの結果を算出するためのいかなる追加の情報も必要としなかった。計算値は、表現の計算値を用いる場合、実際の値を特定しなければならない名付けられたパラメータのリストを宣言することができる。以下は、パラメータを必要とする計算値の例である。

```
type PointPlus {
 X : Number;
 Y : Number;
 // a calculated value that requires a parameter
 WithinBounds(radius：Number) : Logical {
 X * X + Y * Y <= radius * radius;
}
 InMagicQuadrant(){IsHigh && X > 0;}
 IsHigh() : Logical {Y>0;}
}
```

表現においてこの計算値を用いるために、以下のような２つのパラメータのための値が用意される。

```
({X = 100, Y = 200} : PointPlus) .WithinBounds(50)
```

「WithinBounds」の値を計算すると、Ｄは、値５０をシンボルの範囲にバインドする。これは、「WithinBounds」計算値の結果が偽の場合を生じさせる。

【０１２５】

フィールドについての計算値及びデフォルト値の両方が型の定義の一部であって、型に一致する値の一部でないということがＤについて注目される。例えば、これらの３つの型の定義を検討する。

```
type Point {
 X : Number;
 Y : Number;
}
type RichPoint {
 X : Number;
 Y : Number;
 Z =-1： Number;
 IsHigh() : Logical {X < Y;}
}
type WeirdPoint {
 X : Number;
 Y : Number;
 Z = 42 : Number;
 IsHigh() : Logical {false;}
}
```

RichPoint及びWeirdPointだけが２つの必須のフィールド（Ｘ及びＹ）を有するので、以下を記述することができる。

```
{X = 1, Y = 2) }  in RichPoint
{X = 1, Y = 2 }  in WeirdPoint
```

しかしながら、「IsHigh」の計算値は、これらの２つの型のうちの１つがエンティティ値のものとされるときに、利用できるだけである。

({X = 1, Y = 2} : RichPoint) .IsHigh == true
({X = 1, Y = 2} : WeirdPoint) .IsHigh == false
計算値は純粋に型の一部であって値ではないので、以下のように帰属が連鎖されると、
(({X = 1, Y = 2} : RichPoint) : WeirdPoint) .IsHigh == false
最外部の帰属が、どの機能が呼び出されるかについて決定する。
【０１２６】
同様の原理は、デフォルトの値が機能する方法に関して効果を現す。デフォルトの値が型の一部であってエンティティの値でない点が再び注目される。よって、以下の表現が書かれると、
({X = 1, Y = 2} : RichPoint) .Z == -1
基本的なエンティティ値はやはり２つのフィールド値（Ｘ及びＹについてそれぞれ１及び２）を含むだけである。この点について、デフォルトの値が計算値と異なる帰属が連鎖される。例えば、以下の表現を検討する。
(({X = 1, Y = 2 } : RichPoint) : WeirdPoint) . Z == -1
「RichPoint」帰属が最初に適用されるので、得られるエンティティは-1の値を有するZと名付けられたフィールドを有する。しかしながら、値に割り当てられる格納部分が存在しない。即ち、それは値の型の解釈の一部である。従って、「WeirdPoint」帰属があてはまると、それは第１の帰属の結果にあてはまる。第１の帰属は、値がＺについての値を特定するために用いることができるように、Ｚと名付けられたフィールドを有する。よって、「WeirdPoint」で特定されるデフォルトの値は、必要とされない。
【０１２７】
全ての型のように、コンストレイントは、「where」演算子を用いてエンティティ型に適用することができる。以下のＤ型の定義を検討する。

```
Type HighPoint {
 X : Number;
 Y : Number;
} where X < Y;
```

この例では、型「HighPoint」に一致する全ての値は、Ｙ値未満であるＸ値を有することが保証される。これは、以下の表現を意味する。
{X = 100, Y = 200} in HighPoint
!({X = 300, Y = 200) }) in HighPoint)
両方の結果は真である。
【０１２８】
更に、以下の型の定義に関して、

```
type Point {
 X : Number;
 Y : Number;
}
 type Visual {
 Opacity : Number;
}
type VisualPoint {
 DotSize : Number;
} where value in Point && value in Visual;
```

第３の型「VisualPoint」は、少なくとも数のフィールドＸ、Ｙ、Opacity及びDotSizeを有するエンティティの値のセットを名付ける。
【０１２９】
構成することができるより小さいピースへメンバーの宣言をファクタリングすることは共通の欲求であるので、Ｄはまたファクタリングについての明示的な構文サポートを用意する。例えば、「VisualPoint」型の定義は、その構文を用いて書き直すことができる。

```
type VisualPoint : Point, Visual {
 DotSize : Number;
}
```

明確にするために、これは、コンストレイント表現を用いた上記の普通の書き方の定義についての縮めた表現である。更に、この縮めた表現の定義及び普通の書き方の定義の両方は、この普通の書き方の定義に対してさえも等価である。

```
type VisualPoint = {
 X : Number;
 Y : Number;
 Opacity : Number;
 DotSize : Number;
}
```

再び、型の名前は、まさに型を参照する方法である。値自体は、これらを説明するために用いられる型の名前の記録を有していない。

【０１３０】

Ｄはまた、いくつかの特徴を有するＬＩＮＱクエリの理解を拡大して簡単なクエリを書くのをより簡潔にすることができる。キーワードである「where」及び「select」は、バイナリの中置演算子として利用可能である。また、インデクサーは、しっかりと定型化されたコレクションに自動的に追加される。これらの特徴によって、共通のクエリが以下に示すようによりコンパクトに書かれるようになる。

【０１３１】

中間演算子の例として、以下のクエリは、「People」の定義済みのコレクションから３０歳以下の人々を抽出する。

```
from p in People
Where p.Age = 30
Select p
```

等価のクエリは、

```
People where value.Age = 30
```

と書くことができる。

【０１３２】

「where」演算子は、左側のコレクション及び右側のブール表現を取る。「where」演算子は、ブール表現のスコープの中へキーワードの識別子の値を導き、ブール表現は、コレクションの各メンバーにバインドされる。得られるコレクションは、表現が真であるメンバーを含む。よって、表現：

```
Collection where Expression
```

は、

```
from value in Collection
where Expression
Select value
```

に等価である。

【０１３３】

Ｄコンパイラは、強く定型化された要素を用いてコレクションにインデクサーメンバーを加える。例えば、コレクション「People」について、コンパイラは、「FiTSt(TeXt)」、「Last(Text)」及び「Age(Number)」についてのインデクサーを加えることができる。

【０１３４】

従って、記述は、

```
Collection .Field (Expression)
```

は、

```
from value in Collection
where Field == Expression
```

select value
に等しい。
フィールド== Expressionが値を選ぶCollectionの価値から
「select」はまた、中置演算子として利用可能である。以下の簡単なクエリに関して、
from p in People
select p.First + p.Last
「select」表現は、コレクションの各メンバーを介して計算され、結果を返す。中置の「select」を用いると、クエリは、
People select value.First + value.Last
と等価に書くことができる。

【０１３５】
「select」演算子は、左側のコレクション及び右側の任意の表現をとる。「where」と同様に、「select」は、コレクションの各要素に及ぶキーワード識別子の値を導く。「select」演算子は、コレクションの各要素を介して表現をマッピングし、結果を返す。別の例について、記述は、
Collection select Expression
は、以下に等価である。
from value in Collection
select Expression
「select」演算子のささいな利用は、単一のフィールドを抽出することである。
People select value.First
コンパイラは、単一のフィールドを「People.First」及び「People.Last」として直接抽出することができるように、アクセス機構をコレクションに追加する。

【０１３６】
適法なＤ文書を書くために、全てのソーステキストは、モジュールの定義の内容に現れる。モジュールは、定義されているあらゆる型の名前についての上位の名前空間を定義する。モジュールはまた、実際の値及び計算値を格納する範囲を定義するためのスコープを定義する。

【０１３７】
以下は、モジュールの定義の簡単な例である。

```
module Geometry {
 // declare a type
  type Point {
  X： Integer; Y ; Integer;
}
 // declare some extents
 Points : Point*;
 Origin : Point;
 // declare a calculated value
 TotalPointCount {Points.Count + 1;}
}
```

この例では、モジュールは、「Geometry.Point」と名付けられた１つの型を定義する。この型は、ポイントの型が何のように見えるかを説明するが、これらの値を格納することができるあらゆる位置を定義しない。

【０１３８】
この例も、２つのモジュールスコーピングされたフィールド（Point及びOrigin）を含む。モジュールスコーピングされたフィールド宣言は、エンティティ型において用いられるこれらと構文において同一である。しかしながら、一旦範囲が決定されるならば、エンティティ型において宣言されるフィールドは単に、格納場所についての可能性に名付ける。対照的に、モジュールスコープにおいて宣言されるフィールドは、実際の格納場所に名

前を付け、格納場所は、モジュールをロードして解釈するために実施によってマッピングされなければならない。

【０１３９】

更に、モジュールは、インポート指示を用いることによって他のモジュールにおける宣言を参照して、参照された宣言を含むモジュールを名付けることができる。他のモジュールによって参照される宣言について、宣言はエクスポート指示を用いて明示的にエクスポートされる。

【０１４０】

例えば、以下のモジュールを考えると、

```
module MyModule {
 import HerModule; //declares HerType
 export My Type1;
 export MyExtentl;
 type MyType1：Logical*;
 type MyType2：HerType;
 MyExtentl：Number*;
 MyExtent2：HerType;
}
```

「MyTypel」及び「MyExtentl」だけが他のモジュールに見え、これは適法な「HerModule」についての以下の定義を作成する点が注意される。

```
module HerModule {
 import MyModule; // declares MyType1 and MyExtent1
 export HerType;
 type HerType： Text where value.Count< 100;
 type Private：Number where ! (value in MyExtentl) ;
 SomeStorage：MyTypel;
}
```

この例が示すように、モジュールは巡回する依存関係を有する。

【０１４１】

Ｄ言語の型は、２つの主なカテゴリに分割される。即ち、固有型及び派生型である。固有型は、Ｄ言語のコンストラクトを用いて定義することができないがむしろＤ言語の仕様書において完全に定義される型である。固有型は、その仕様書の一部としてその上位型として多くても１つの固有型に名前を付けることができる。値は、正確に１つの固有型の例であって、その１つの固有型の明細書及びその上位型の全てに一致する。

【０１４２】

派生型は、その定義を、言語内に用意されかつ型コンストラクタを用いてＤソーステキストにおいて構築することができる型である。派生型は、別の型を介してコンストレイントとして定義される。それは、明示的なサブタイプの関係を生成する。値は、単に派生型のコンストレイントを満たすことに基づいていかなる数の派生型に一致する。値と派生型との間の演繹的な所属関係は存在しない。むしろ、派生型のコンストレイントに一致する所定の値は、自由にその型と解釈することができる。

【０１４３】

Ｄは型を定義する際のオプションの幅広い範囲を提供する。コレクションを返す表現は型と宣言することができる。エンティティ及びコレクションについての型述部は表現であって、この形式に適合する。型の宣言は明示的にそのメンバーを列挙してもよいしまたは他の型から成っていてもよい。

【０１４４】

Ｄのような構造的に定型化された言語と名目上定型化された言語との間に別の区別がある。Ｄにおける型は、値のセットについての仕様書である。２つの型は、値の正確な同じコレクションが型の名前に関係なく両方ともに一致する場合に同じものである。名付けら

れた型が用いられる必要はない。どこで型の参照が必要とされても型の表現は可能である。Ｄにおける型は単に、コレクションを返す表現である。

【０１４５】

型Ａに一致するあらゆる値はまた型Ｂに一致し、次にＡはＢのサブタイプである（かつＢはＡの上位型である）。サブタイプは、推移的である。即ち、ＡはＢのサブタイプであり、ＢはＣのサブタイプである場合、ＡはＣのサブタイプである（かつＣはＡの上位型である）。サブタイプは、再帰的である。即ち、ＡはＡの（無効の）サブタイプである（かつＡはＡの上位型である）。

【０１４６】

型は、型述部を満たす全ての値のコレクションと考えられる。従って、コレクションに関するいかなるオペレーションも型に適用することができ、型はいずれかの他のコレクションの値のような表現で操作することができる。

【０１４７】

Ｄは、値が出現する２つの主な手段を設けている。即ち、計算値及び格納値（別名、フィールド）。計算値及び格納値は、モジュール及びエンティティ宣言の両方で生じることができて、これらのコンテナによってスコーピングされる。計算値は、Ｄソースのテキストの一部として通常定義される表現を評価することから派生される。対照的に、フィールドは値を格納し、フィールドのコンテンツは経時変化することができる。

【０１４８】

（例示的なネットワーク化された環境及び分散された環境）

当業者は、本明細書において説明される宣言型プログラミングモデルについての有向グラフ構造についての様々な実施形態が、いかなるコンピューターまたは他のクライアントデバイスもしくはサーバーデバイスと関連して実行することができると認めることができる。これらはコンピューターネットワークの一部としてまたは分散コンピューティング環境において配備することができ、あらゆる種類のデータストアに接続することができる。この点について、本明細書において説明されている様々な実施形態は、あらゆるメモリまたはストレージユニット並びにあらゆる数のストレージユニットにわたって生じるあらゆるアプリケーション及びプロセスを有するあらゆるコンピューターシステムまたはコンピューター環境において実施することができる。これは、リモートまたはローカルのストレージを有するネットワーク環境または分散コンピューティング環境に配置されるサーバーコンピューター及びクライアントコンピューターを含むが、これらに限定されるものではない。

【０１４９】

分散コンピューティングは、コンピューティングデバイス及びコンピューティングシステムの中のコミュニケーションの交換によってコンピューターのリソース及びサービスを共有することを提供する。これらのリソース及びサービスは、ファイルなどの情報、キャッシュストレージ及びオブジェクトについてのディスクストレージの交換を含む。これらのリソース及びサービスはまた、ロードバランシング、リソースの拡大、処理の特殊化などのために複数の処理ユニットにわたって処理能力を共有することを含む。分散コンピューティングはネットワークの接続性を利用し、クライアントが企業全体のためになるこれらの総体的な力を活用できるようにする。この点について、様々なデバイスは、アプリケーション、オブジェクト及びリソースを有することができ、これらは、本発明の開示についての様々な実施形態のうちのいずれかの１つまたは複数の態様を共同して実行することができる。

【０１５０】

図２４は、例示的なネットワーク化されたコンピューティング環境または分散されたコンピューティング環境の概略図を提供する。分散コンピューティング環境は、コンピューティングオブジェクト２４１０、２４１２等及びコンピューティングオブジェクトまたはコンピューティングデバイス２４２０、２４２２、２４２４、２４２６、２４２８等を含む。これらは、アプリケーション２４３０、２４３２、２４３４、２４３６、２４３８で

表されるような、プログラム、方法、データストア、プログラマブル論理等を含むことができる。オブジェクト２４１０、２４１２等及びコンピューティングオブジェクトまたはコンピューティングデバイス２４２０、２４２２、２４２４、２４２６、２４２８等は、例えばＰＤＡ、オーディオデバイス／ビデオデバイス、携帯電話、ＭＰ３プレーヤ、パーソナルコンピューター、ラップトップ等などの異なったデバイスを含むことができる。

【０１５１】

各オブジェクト２４１０、２４１２等、コンピューティングオブジェクトまたはコンピューティングデバイス２４２０、２４２２、２４２４、２４２６、２４２８等は、通信ネットワーク２４４０を経由して直接または間接的に、１つまたは複数の他のオブジェクト２４１０、２４１２等及びコンピューティングオブジェクトまたはコンピューティングデバイス２４２０、２４２２、２４２４、２４２６、２４２８と通信することができる。図２４に示した単一の要素として示されているが、ネットワーク２４４０は、図２４のシステムにサービスを提供する他のコンピューティングオブジェクト及びコンピューティングデバイスを含むことができ、かつ／または示されていない複数の相互接続されるネットワークを表すことができる。各オブジェクト２４１０、２４１２等または２４２０、２４２２、２４２４、２４２６、２４２８等も、ＡＰＩを使用することができるアプリケーション２４３０、２４３２、２４３４、２４３６、２４３８などのアプリケーション、または本発明の開示の様々な実施形態に基づいて提供される宣言型プログラミングモデルについての有向グラフ構造との通信、このために処理、またはこれの実施に適した、他のオブジェクト、ソフトウェア、ファームウェア及び／またはハードウェアを含むことができる。

【０１５２】

分散コンピューティング環境をサポートする様々なシステム、コンポーネント及びネットワーク構成が存在する。例えば、コンピューティングシステムは有線システムもしくは無線システムによって、ローカルネットワークもしくは広範な分散ネットワークで接続することができる。現在、いずれのネットワークインフラストラクチャも、様々な実施例において説明したように、宣言型プログラミングモデルのための使用している有向グラフ構造から起こる例示的な通信のために用いることができるが、多くのネットワークは、インターネットに接続していて、インターネットは、広範に分散されたコンピューティングのためのインフラストラクチャを提供し、多くの異なるネットワークを含む。

【０１５３】

よって、クライアント／サーバー、ピアツーピアまたはハイブリッドアーキテクチャなどの、ネットワークトポロジ及びネットワークインフラストラクチャのホストを利用することができる。「クライアント」は、クライアントが関係していない別のクラスまたはグループのサービスを使用するクラスまたはグループのメンバーである。クライアントは、プロセスであってもよい。即ち、大ざっぱに言って、別のプログラムまたはプロセスが提供するサービスを必要とする、命令またはタスクのセットであってもよい。クライアントのプロセスは、他のプログラムについてのあらゆる機能の詳細またはサービス自体を「知る」必要なしに要求されたサービスを利用する。

【０１５４】

特にネットワーク化されたシステムであるクライアント／サーバーアーキテクチャでは、クライアントは通常、サーバーなどの別のコンピューターによって提供される共有ネットワークリソースにアクセスするコンピューターである。図２４の説明では、非限定的な例として、コンピューター２４２０、２４２２、２４２４、２４２６、２４２８等は、クライアントと考えることができ、コンピューター２４１０、２４１２等は、サーバーと考えることができる。ここで、サーバー２４１０、２４１２等は、環境に基づいていかなるコンピューターもクライアント、サーバー、またはその両方と考えることができるが、クライアントコンピューター２４２０、２４２２、２４２４、２４２６、２４２８からデータを受信すること、データを格納すること、データを処理すること、データをクライアントコンピューター２４２０、２４２２、２４２４、２４２６、２４２８等に送信することなどのデータサービスを提供する。これらのコンピューティングデバイスのいずれも、デ

ータを処理し、データをコード化し、データについてクエリし、または１つもしくは複数の実施形態について本明細書において説明されたような宣言型プログラミングモデルのための有向グラフ構造の処理を関係させることができるサービスまたはタスクを要求することができる。

【０１５５】

サーバーは通常、インターネットまたは無線ネットワークインフラストラクチャなどのリモートのネットワークまたはローカルのネットワークを介してアクセス可能なリモートのコンピューターシステムである。クライアントプロセスが、第１のコンピューターシステムで動作中でもよくかつサーバープロセスが第２のコンピューターシステムで動作中でもよく、通信媒体を介して互いに通信し、よって、分散機能を提供し、複数のクライアントがサーバーの情報収集能力を利用できるようにする。宣言型プログラミングモデルについての有向グラフ構造の処理に従って利用されるいかなるソフトウェアオブジェクトも、独立していてもよいし、または複数のコンピューティングデバイスまたはオブジェクトにわたって分散されて提供されてもよい。

【０１５６】

通信ネットワーク／バス２４４０がインターネットであるネットワーク環境において、例えばサーバー２４１０、２４１２等は、ウェブサーバーであってもよく、ウェブサーバーと、クライアント２４２０、２４２２、２４２４、２４２６、２４２８等は、ハイパーテキスト転送プロトコル（ＨＴＴＰ）などの、多くの公知のプロトコルを介して通信する。サーバー２４１０、２４１２等はまた、分散コンピューティング環境に特有であってもよいような、クライアント２４２０、２４２２、２４２４、２４２６、２４２８等として機能することができる。

【０１５７】

（例示的なコンピューティングデバイス）

上記のように、本明細書において説明されている技術は、素早く大量のデータをクエリすることができるデータ集中的なアプリケーションを作成することが望ましいあらゆるデバイスに適用することができる。従って、ハンドヘルドコンピューティングデバイス、携帯用コンピューティングデバイス及び全ての種類の他のコンピューティングデバイス及び全ての種類のコンピューティングオブジェクトが様々な実施形態、即ち、デバイスが速くかつ効果的な結果のために大量のデータを走査するかまたは処理することを望むことができるどこでも、に関連して仕様を考慮されると理解されなければならない。従って、多目的リモートコンピューターが図２５で後述した以下の汎用のリモートコンピューターは、コンピューティングデバイスの１つの例にすぎない。

【０１５８】

必要ではないが、実施形態は部分的に、デバイスまたはオブジェクトのためのサービスの開発者が使用するために、オペレーティングシステムを介して実行することができかつ／または本明細書において説明されている様々な実施形態のうちの１つまたは複数の機能態様を実行するよう動作するアプリケーションソフトウェア内に含むことができる。ソフトウェアは、クライアントワークステーション、サーバーまたは他のデバイスなどの１つまたは複数のコンピューターが実行するプログラムモジュールなどのコンピューター実行可能命令の一般的な内容で説明することができる。当業者は、通信データに用いることができる様々な構成及びプロトコルを有すると認めることができ、よって、特定のコンフィギュレーションまたはプロトコルは限定するとみなされてはならない。

【０１５９】

よって図２５は、本明細書において説明されている実施形態のうちの１つまたは複数の態様を実施することができる適切なコンピューターシステム環境２５００の例を示す。但し、上記で明らかにされたように、コンピューターシステム環境２５００は適切なコンピューティング環境の１つの例にすぎず、使用または機能の範囲に関していかなる限定も示唆することを意図としていない。コンピューティング環境２５００は例示的な動作環境２５００において示されたコンポーネントのうちのいずれか１つまたは組合せに関するいか

なる依存性または要件も有すると解釈されてはならない。

【０１６０】

図２５を参照すると、１つまたは複数の実施形態を実施するための例示的なリモートデバイスは、コンピューター２５１０の形で汎用コンピューティングデバイスを含む。コンピューター２５１０のコンポーネントは、処理ユニット２５２０、システムメモリ２５３０及び処理装置２５２０に対するシステムメモリを含む様々なシステムコンポーネントを接続するシステムバス２５２２を含むがこれらに限定されるものではない。

【０１６１】

コンピューター２５１０は通常、様々なコンピューター可読媒体を含み、コンピューター２５１０がアクセスすることができるあらゆる利用可能な媒体であってもよい。システムメモリ２５３０は、リードオンリメモリ（ＲＯＭ）及び／またはランダムアクセスメモリ（ＲＡＭ）などの揮発性メモリ及び／または不揮発性メモリの形でコンピューター記憶媒体を含むことができる。例であって限定ではないが、メモリ２５３０はまた、オペレーティングシステム、アプリケーションプログラム、他のプログラムモジュール及びプログラムデータを含むことができる。

【０１６２】

ユーザは、入力デバイス２５４０を介してコンピューター２５１０に、コマンド及び情報を入力することができる。モニタまたは他のタイプのディスプレイデバイスはまた、出力インターフェース２５５０などのインターフェースを介してシステムバス２５２２に接続されている。モニタに加えて、コンピューターはまた、出力インターフェース２５５０を介して接続することができる、スピーカ及びプリンタなどの他の周辺出力装置を含むことができる。

【０１６３】

コンピューター２５１０は、リモートコンピューター２５７０などの１つまたは複数の他のリモートコンピューターへの論理接続を用いてネットワーク環境または分散環境において動作することができる。リモートコンピューター２５７０は、パーソナルコンピューター、サーバー、ルータ、ネットワークＰＣ、ピアデバイスもしくは他の共通ネットワークノード、またはあらゆる他のリモート媒体消費デバイスもしくは伝送デバイスであってもよくて、コンピューター２５１０と関連して上記の要素のうちのいずれかまたは全てを含むことができる。図２５において示した論理接続は、かかるローカルエリアネットワーク（ＬＡＮ）またはワイドエリアネットワーク（ＷＡＮ）などのネットワーク２５７２を含むが、他のネットワーク／バスを含むことができる。かかるネットワーク環境は、家庭、オフィス、企業規模のコンピューターネットワーク、イントラネット及びインターネットにおいて普通のことである。

【０１６４】

上記のように、例示的な実施形態が様々なコンピューティングデバイス及びネットワークアーキテクチャと関連して説明されてきたが、基本的な概念は、いかなるネットワークシステムにもかついかなるコンピューティングデバイス及びコンピューティングシステムにも適用することができる。そこにおいて、例えば、大規模なデータを介してクエリを処理する環境では、マシンまたは人間の読み込み可能なフォーマットで直接データ集中的なアプリケーションを作成することが望ましい。

【０１６５】

また、アプリケーション及びサービスが宣言型プログラミングモデルについての有向グラフ構造を用いるのを可能にする、適当なＡＰＩ、ツールキット、ドライバコード、オペレーティングシステム、制御、独立型または、ダウンロード可能なソフトウェアオブジェクト等などの同じ機能または類似の機能を実施する複数の方法が存在する。よって、本明細書の実施形態は、ＡＰＩ（または他のソフトウェアオブジェクト）の見地からかつ宣言型プログラミングモデルのための有向グラフ構造を提供し、作成し、処理しまたは格納するソフトウェアオブジェクトまたはハードウェアオブジェクトから考慮される。よって、本明細書において説明されている様々な実施形態は、ソフトウェアにおいてと同様に、完

全にハードウェアにおける態様、部分的にハードウェアにおける態様、かつ部分的にソフトウェアにおける態様及びソフトウェアにおける態様を有することができる。

【０１６６】

文言「例示的な」は、本明細書において例、実施例または説明として役立つことを意味するために用いられる。誤解を避けるために、本明細書において開示される本発明は、かかる例によって限定されない。更に、「例示的」と本明細書において説明されているあらゆる態様及びデザインは、必ずしも他の態様またはデザインに勝って好適であるかまたは利点があると解釈される必要はないし、また、当業者に公知の対応する例示的な構造及び技術を除外することを意味しない。更に、用語「含む」、「有する」、「備える」及び他の同様の文言が、詳細な説明または特許請求の範囲のいずれかにおいて用いられる範囲まで、誤解を避けるために、かかる用語は、あらゆる追加の要素または他の要素を除外することなしに率直な移行ことばとして、用語「含む」と同様の方法で包括的あることを意図している。

【０１６７】

上記のように、本明細書において説明した様々な技術は、ハードウェアもしくはソフトウェア、または、必要に応じて、両方の組合せと関連して実施することができる。本明細書において用いられるように、用語「コンポーネント」、「システム」などは同様に、コンピューター関連のエンティティ、ハードウェア、ハードウェア及びソフトウェアの組合せ、ソフトウェア、または実行中のソフトウェアのいずれかを言うと意図されている。例えば、コンポーネントは以下のものであってもよいが、これらであると限定されない。以下のものは、プロセッサで実行中のプロセス、プロセッサ、オブジェクト、実行ファイル、実行スレッド、プログラム、及び／またはコンピューターである。実例として、コンピューターで実行中のアプリケーション及びコンピューターは、コンポーネントであってもよい。１つまたは複数のコンポーネントは、プロセス及び／または実行スレッド内に存在し、コンポーネントは、１つのコンピューターで局所化されてもよいし、かつ／または２つ以上のコンピューターの間で分散されてもよい。

【０１６８】

上述のシステムは、いくつかのコンポーネント同士の間のインタラクションに関して説明された。かかるシステム及びコンポーネントは、これらのコンポーネントまたは特定のサブコンポーネント、特定のコンポーネントまたはサブコンポーネントのうちのいくつか、及び／または追加のコンポーネントを含むことができ、前述のものの様々な順列及び組合せに基づくことができると認められる。サブコンポーネントはまた、親コンポーネント（階層的な）内に含まれるよりはむしろ、他のコンポーネントにコミュニケーション可能に接続されたコンポーネントとして実装することができる。更に、１つまたは複数のコンポーネントは、集積された機能を提供している単一のコンポーネントに組み合わされてもよいしまたはいくつかの別個のサブコンポーネントに分割されてもよく、管理層などのいずれかの１つまたは複数の中間層は、統合した機能を提供するためにかかるサブコンポーネントに通信可能に接続するために提供することができる点が注目されるべきである。本明細書において説明されているあらゆるコンポーネントはまた、本明細書において具体的に説明されていないが、当業者一般に公知の１つまたは複数の他のコンポーネントとインタラクションすることができる。

【０１６９】

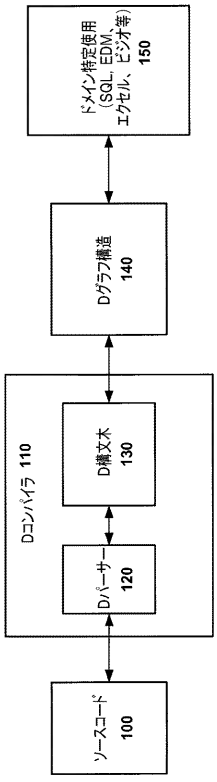
上記に説明された例示的なシステムを考慮して、説明されている本発明に基づいて実施することができる方法は、様々な図のフローチャートを参照するとよりよく理解される。説明を簡易にするめに、方法が一連のブロックとして示されかつ説明されているが、いくつかのブロックは、本明細書において記載されているものと異なる順序でかつ／または他のブロックと並行して起きてもよいように、請求された本発明はブロックの順序で限定されないと理解されかつ認められることになっている。非逐次的または分岐されたフローがフローチャートを介して示されているが、同じ結果または類似する結果を実現する、様々な他の分岐、フロー経路及びブロックの順序を実行することができると認められる。更に

、全ての示したブロックを、以下に説明される方法を実施するために必要とするわけでない。

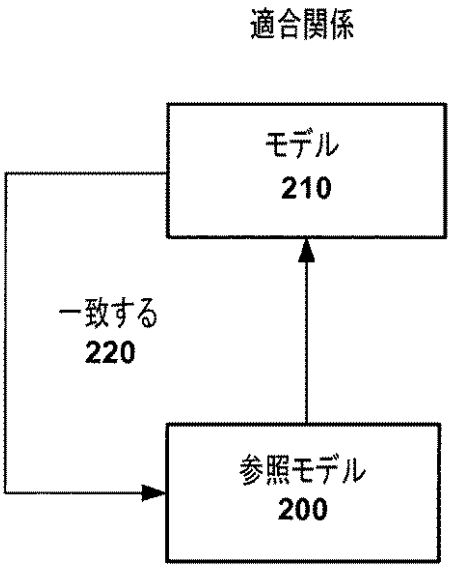
【０１７０】

本明細書において説明されている様々な実施形態に加えて、他の類似する実施形態を用いることができるかまたは、変更及び追加が、そこから逸脱することなく対応する実施形態（または複数の実施形態）の同じ機能または均等の機能を実行するために説明されている実施形態（または複数の実施形態）に行うことができると理解されることになっている。また更に、複数の処理チップまたは複数のデバイスは、本明細書において説明されている１つまたは複数の機能の性能を共有することができ、同様に、ストレージは複数のデバイスにわたって達成することができる。従って、本発明はいずれかの単一の実施形態に限定されるべきではなく、むしろ添付の特許請求の範囲に基づく広さ、趣旨及びスコープにおいて解釈されなければならない、。

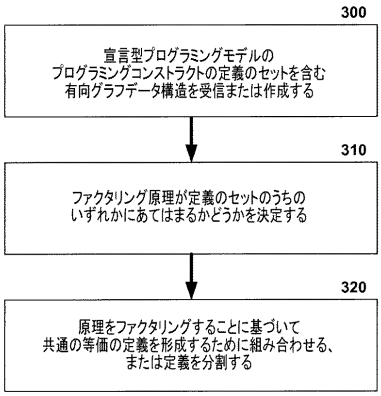
【図１】



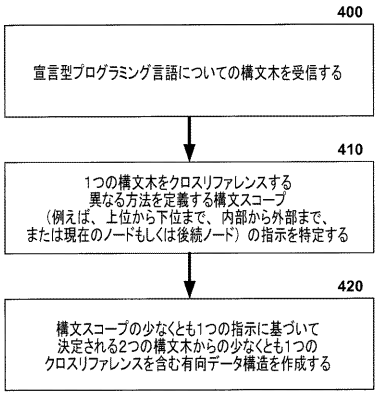
【図２】



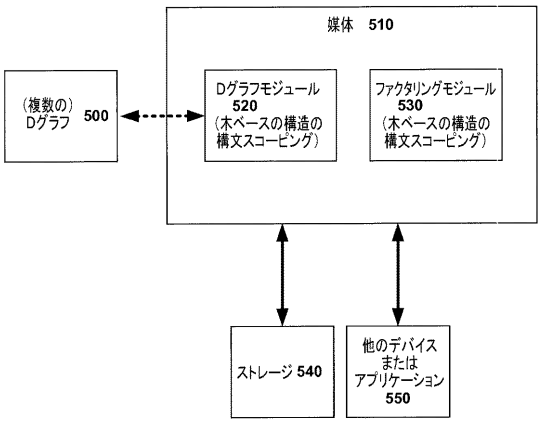
【図３】



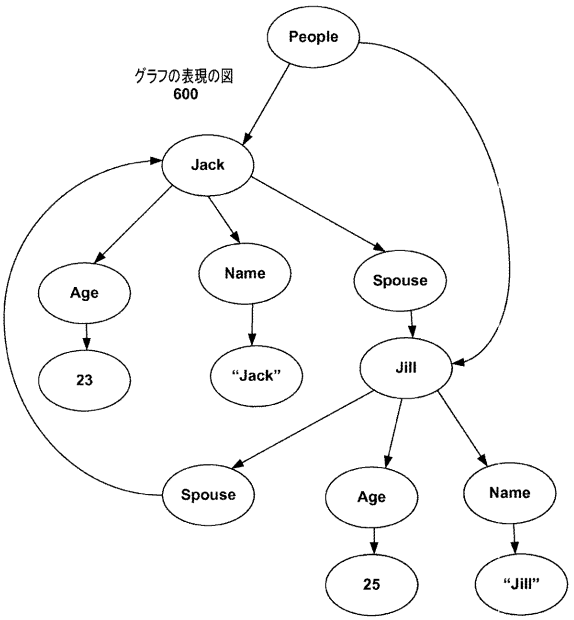
【図４】



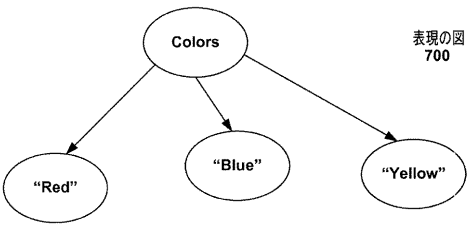
【図５】



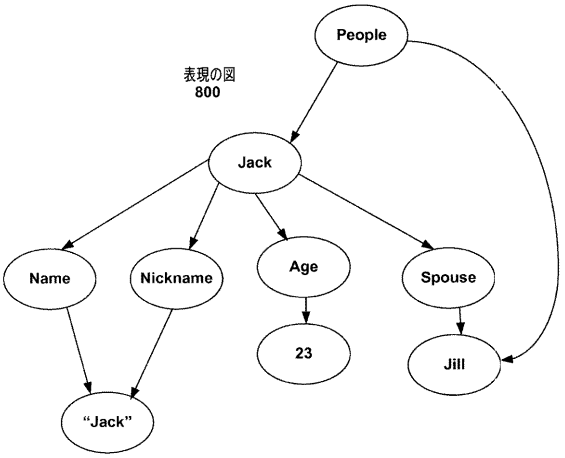
【図６】



【図７】



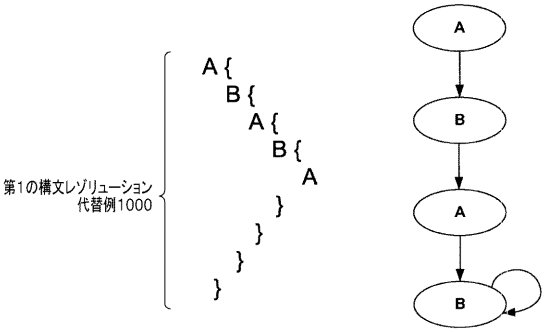
【図８】



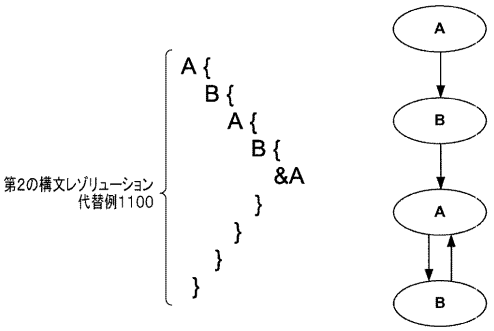
【図９】

2つの選択	内側から外側	上部から
ノード	代替例1000（例えば、図10）	代替例1200（例えば、図12）
後続処理	代替例1100（例えば、図11）	代替例1300（例えば、図13）

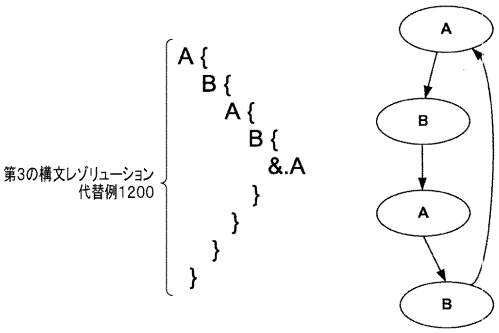
【図１０】



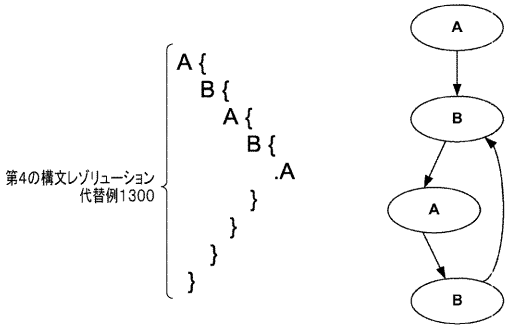
【図１１】



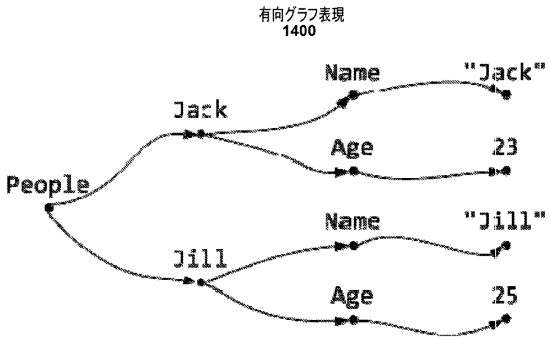
【図１２】



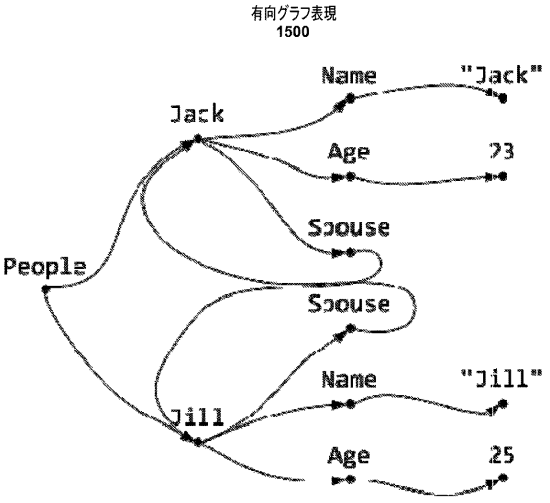
【図１３】



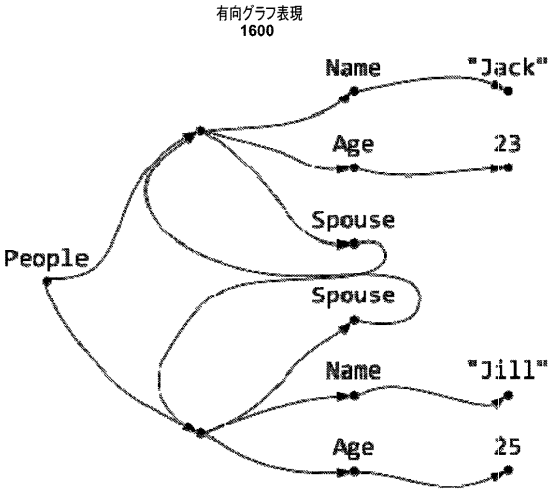
【図１４】



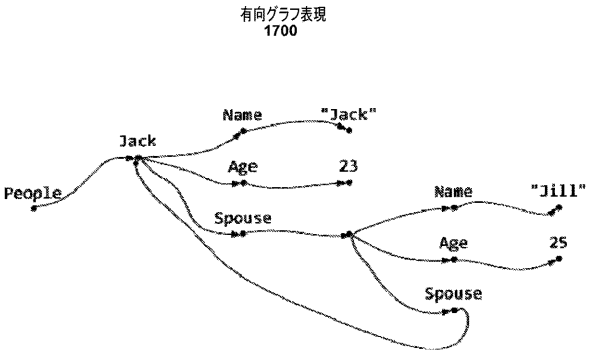
【図１５】



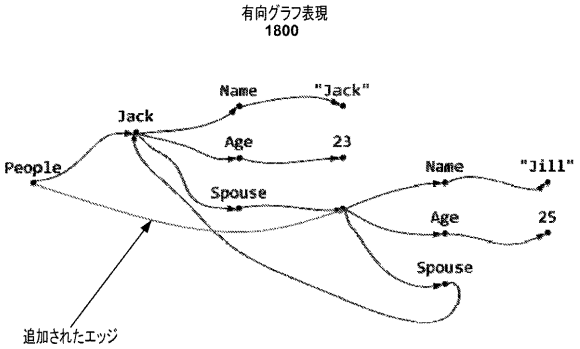
【図１６】



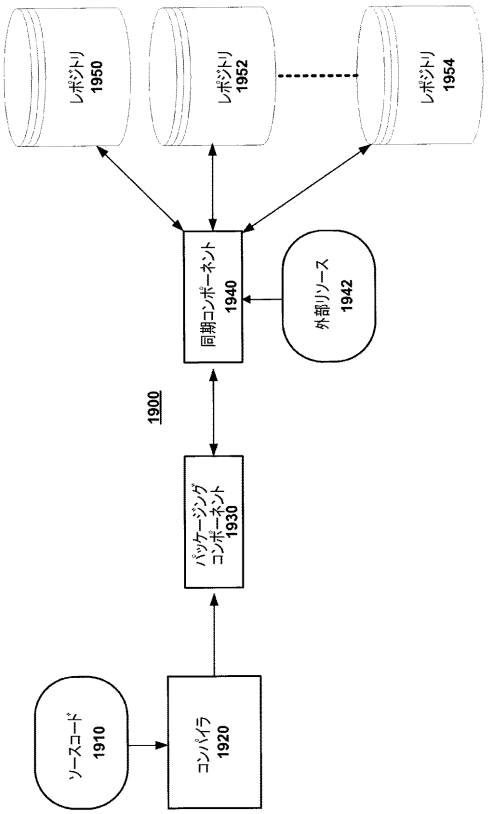
【図１７】



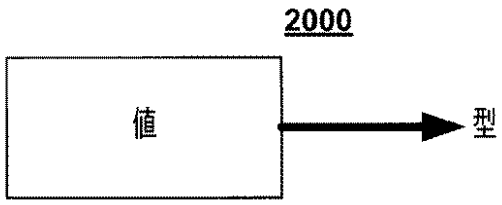
【図１８】



【図１９】

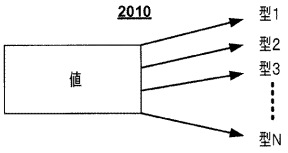


【図２０】



従来技術

【図２１】



【図２２】

2200

順序	Bar値
1	A
2	B
3	C

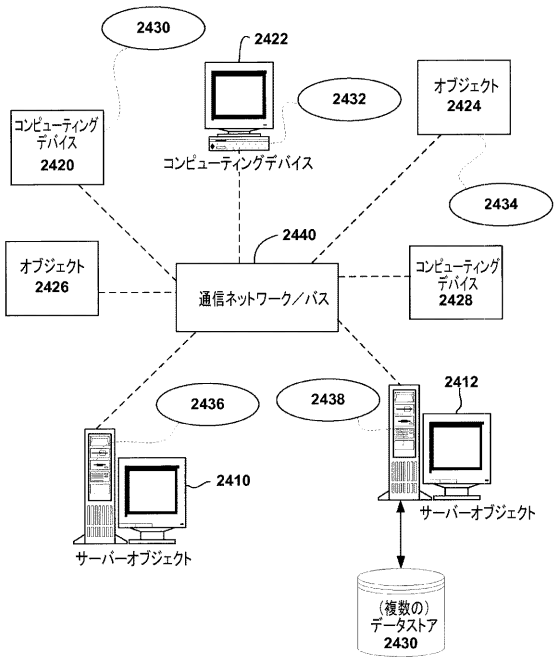
【図２３】

2210

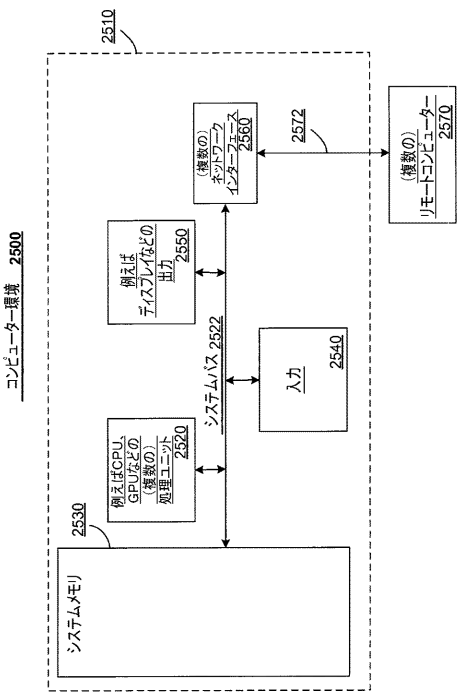
ID	Bar値
XXX	A
YYY	B
ZZZ	C

従来技術

【図２４】



【図２５】



フロントページの続き

(74)代理人 100120112
弁理士 中西 基晴
(74)代理人 100147991
弁理士 鳥居 健一
(74)代理人 100119781
弁理士 中村 彰吾
(74)代理人 100162846
弁理士 大牧 綾子
(74)代理人 100173565
弁理士 末松 亮太
(74)代理人 100138759
弁理士 大房 直樹
(74)代理人 100091063
弁理士 田中 英夫
(72)発明者 デイビッド イー．ラングワーシー
アメリカ合衆国 ９８０５２ ワシントン州 レッドモンド ワン マイクロソフト ウェイ マイクロソフト コーポレーション エルシーエー－インターナショナル パテンツ内
(72)発明者 ジョン エル．ハンビー
アメリカ合衆国 ９８０５２ ワシントン州 レッドモンド ワン マイクロソフト ウェイ マイクロソフト コーポレーション エルシーエー－インターナショナル パテンツ内
(72)発明者 ブラッドフォード エイチ．ラブリング
アメリカ合衆国 ９８０５２ ワシントン州 レッドモンド ワン マイクロソフト ウェイ マイクロソフト コーポレーション エルシーエー－インターナショナル パテンツ内
(72)発明者 ドナルド エフ．ボックス
アメリカ合衆国 ９８０５２ ワシントン州 レッドモンド ワン マイクロソフト ウェイ マイクロソフト コーポレーション エルシーエー－インターナショナル パテンツ内

審査官 塚田 肇

(56)参考文献 ｕＣｏｓｍｉｎｅｘｕｓ ＸＭＬ Ｌｉｎｋ Ｐｒｏｃｅｓｓｏｒユーザーズガイド，株式会社日立製作所，２００５年 7月，第1版，ｐｐ．１７－２１
Serge Abiteboul，XMLデータベース入門，２００６年 7月１５日，第1版，ｐｐ．３４－３８，１６１

(58)調査した分野(Int.Cl.，ＤＢ名)
Ｇ０６Ｆ ９／４４