Office de la Propriété Intellectuelle du Canada
Un organisme d'Industrie Canada

Canadian Intellectual Property Office
An agency of Industry Canada

CA 2099918 C 2002/11/19

(11)(21) 2 099 918

(12) BREVET CANADIEN
CANADIAN PATENT

(13) C

(22) Date de dépôt/Filing Date: 1993/07/06

(41) Mise à la disp. pub./Open to Public Insp.: 1994/01/07

(45) Date de délivrance/Issue Date: 2002/11/19

(30) Priorité/Priority: 1992/07/06 (07/909,983) US

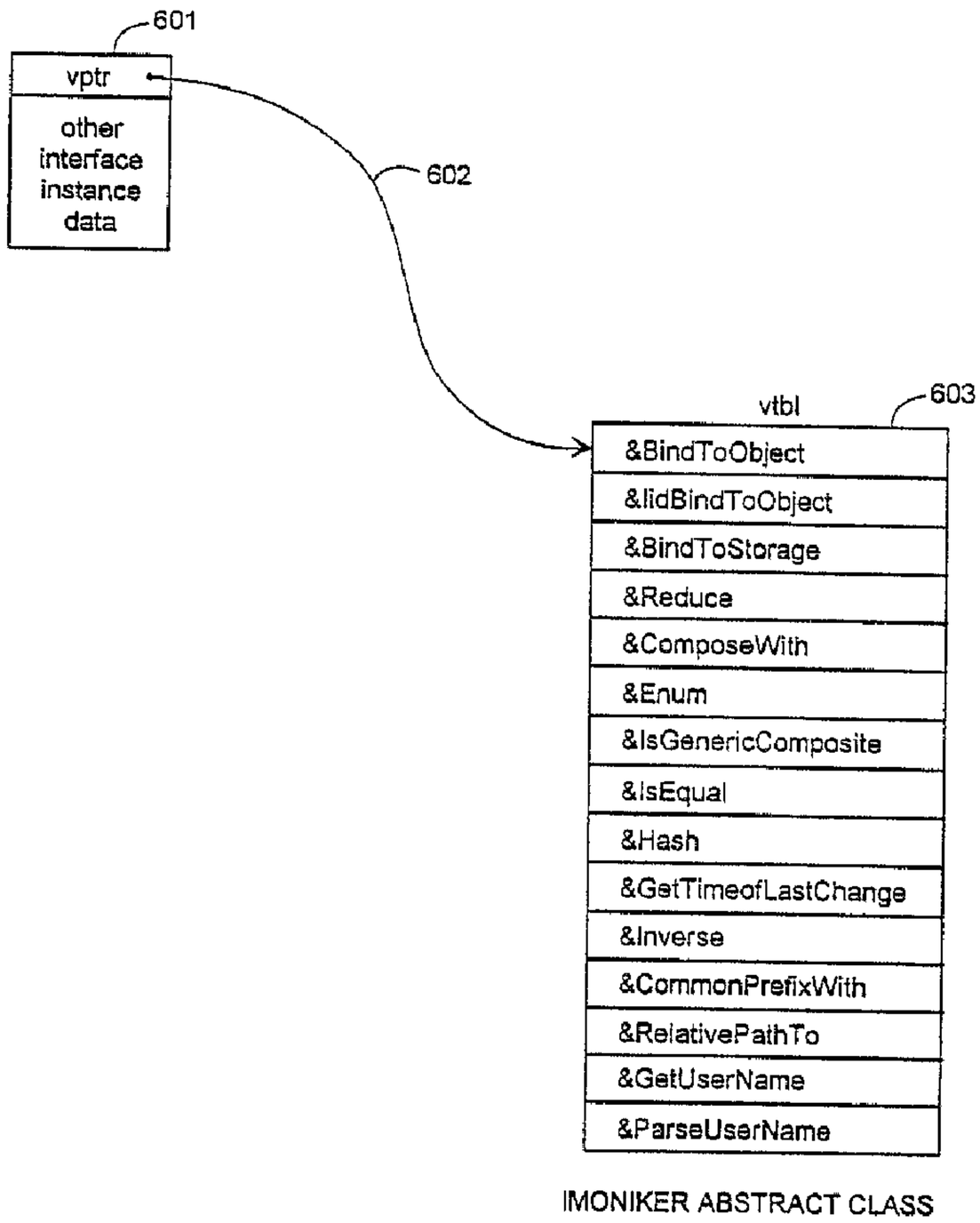
(51) Cl.Int.^{5}/Int.Cl.^{5} G06F 15/40

(72) Inventeurs/Inventors:
Atkinson, Robert G., US;
Williams, Antony S., US;
Jung, Edward K., US

(73) Propriétaire/Owner:
MICROSOFT CORPORATION, US

(74) Agent: OYEN WIGGS GREEN & MUTALA

(54) Titre : METHODE ET SYSTEME POUR LE NOMMAGE ET L'EDITION DE LIENS
(54) Title: METHOD AND SYSTEM FOR NAMING AND BINDING OBJECTS



(57) Abrégé/Abstract:
A method and system for naming and binding data objects. In a preferred embodiment, a link to an object incorporated is stored as a moniker. A moniker is an object that encapsulates the information needed to access the incorporated data and provides methods which bind to the incorporated data.

METHOD AND SYSTEM FOR NAMING AND BINDING OBJECTS

Abstract of the Disclosure

A method and system for naming and binding data objects. In a preferred embodiment, a link to an object incorporated is stored as a moniker. A moniker is an object that encapsulates the information needed to access the incorporated data and provides methods which bind to the incorporated data.

c:\mjp\1296

Description

A METHOD AND SYSTEM FOR NAMING AND BINDING OBJECTS

Technical Field

This invention relates generally to a computer method and system for naming and binding objects and, more specifically, to a generalized method and system for naming linked objects, which may be contained in other objects, locating and binding to the sources of linked objects, and reducing the number of server invocations required to bind an object.

Background of the Invention

Current document processing computer systems allow a user to prepare compound documents. A compound document is a document that contains information in various formats. For example, a compound document may contain data in text format, chart format, numerical format, etc. Figure 1 is an example of a compound document. In this example, the compound document 101 is generated as a report for a certain manufacturing project. The compound document 101 contains scheduling data 102, which is presented in chart format; budgeting data 103, which is presented in spreadsheet format; and explanatory data 104, which is presented in text format. In typical prior systems, a user generates the scheduling data 102 using a project management computer program and the budgeting data 103 using a spreadsheet computer program. After this data has been generated, the user creates the compound document 101, enters the explanatory data 104, and incorporates the scheduling data 102 and budgeting data 103 using a word processing computer program.

Figure 2 shows how the scheduling data, budgeting data, and explanatory data can be incorporated

into the compound document. The user generates scheduling data using the project management program 201 and then stores the data in the clipboard 203. The user generates budgeting data using the spreadsheet program 204 and then stores the data in the clipboard 203. The clipboard 203 is an area of storage (disk or memory) that is typically accessible by any program. The project management program 201 and the spreadsheet program 204 typically store the data into the clipboard in a presentation format. A presentation format is a format in which the data is easily displayed on an output device. For example, the presentation format may be a bitmap that can be displayed with a standard bitmap block transfer operation (BitBlt). The storing of data into a clipboard is referred to as "copying" to the clipboard.

After data has been copied to the clipboard 203, the user starts up the word processing program 206 to create the compound document 101. The user enters the explanatory data 104 and specifies the locations in the compound document 101 to which the scheduling data and budgeting data that are in the clipboard 203 are to be copied. The copying of data from a clipboard to a document is referred to as "pasting" from the clipboard. The word processing program 206 then copies the scheduling data 102 and the budgeting data 103 from the clipboard 203 into the compound document 101 at the specified locations. Data that is copied from the clipboard into a compound document is referred to as "embedded" data. The word processing program 206 treats the embedded data as simple bitmaps that it displays with a BitBlt operation when rendering the compound document 101 on an output device. In some prior systems, a clipboard may only be able to store data for one copy command at a time. In such a system, the scheduling data can be copied to the clipboard and then pasted into the compound document. Then, the budgeting data can be copied to the clipboard and then pasted into the compound document.

Since word processors typically process only text data, users of the word processing program can move or delete embedded data, but cannot modify embedded data, unless the data is in text format. Thus, if a user wants to modify, for example, the budgeting data 103 that is in the compound document 101, the user must start up the spreadsheet program 204, load in the budgeting data 103 from a file, make the modifications, copy the modifications to the clipboard 203, start up the word processing program 206, load in the compound document 101, and paste the modified clipboard data into the compound document 101.

Some prior systems store links to the data to be included in the compound document rather than actually embedding the data. When a word processing program pastes the data from a clipboard into a compound document, a link is stored in the compound document. The link points to the data (typically residing in a file) to be included. These prior systems typically provide links to data in a format that the word processing program recognizes or treats as presentation format. For example, when the word processing program 206 is directed by a user to paste the scheduling data and budgeting data into the compound document by linking, rather than embedding, the names of files in which the scheduling data and budgeting data reside in presentation format are inserted into the document. Several compound documents can contain links to the same data to allow one copy of the data to be shared by several compound documents.

Summary of the Invention

It is an object of the present invention to provide a method and system for generating links to data incorporated within a compound document.

It is another object of the present invention for binding links to the underlying data.

It is another object of the present invention for interfacing with these links in a manner that is independent of the underlying data.

It is another object of the present invention for linking to data nested to an arbitrary level within a compound document.

These and other objects, which will become apparent as the invention is more fully described below, are provided by a method and system for naming and binding data objects. In a preferred embodiment, a link to an object incorporated is stored as a moniker. A moniker is an object that encapsulates the information needed to access the incorporated data and provides methods which bind to the incorporated data.

Brief Description of the Drawings

Figure 1 is an example of a compound document.

Figure 2 shows how the scheduling data, budgeting data, and explanatory data can be incorporated into the compound document.

Figure 3 shows an example of the relationship of a linked object to its source.

Figure 4 is a block diagram of a sample instance of an object.

Figure 5 is a block diagram showing public view of an object.

Figure 6 is a block diagram showing a typical instance of the IMoniker interface.

Figure 7 is a block diagram showing the moniker that is stored in the executive summary compound document 303 in Figure 3.

Figure 8 is a block diagram showing a typical composite moniker object data structure.

Figure 9 is a flow diagram of an implementation of the generic composite moniker method ComposeWith.

Figure 10 is a flow diagram for the function MkCreateGenericComposite.

Figure 11 is a flow diagram of a FindLastElt method of a typical implementation. This method creates two monikers.

Figure 12 is a flow diagram for the FindFirstElt method of a typical implementation.

Figure 13 is a flow diagram of the BindToObject method for a typical implementation.

Figure 14 is a flow diagram of the BindToObject method for a typical implementation of a file moniker.

Figure 15 is a flow diagram of the BindToObject method for a typical implementation of an item moniker.

Figure 16 is a flow diagram of the BindToStorage method for a typical implementation of WItemMoniker.

Figure 17 shows an overview flow diagram for the reduce method for the generic composite moniker interface.

Figure 18 shows an overview flow diagram of the Reduce method for a typical implementation of a file moniker.

Figure 19 shows an overview flow diagram of the Reduce method for a typical implementation of the word processing item moniker WItemMoniker.

Figure 20 shows an overview flow diagram for the function MkParseUserName.

Figure 21 shows an overview flow diagram for the method ParseUserName for a typical implementation of a file moniker.

Figure 22 is a diagram illustrating a linked object.

Detailed Description of the Invention

The present invention provides a generalized method for naming and binding to linked data. In a preferred embodiment, a compound document that incorporates linked data stores a persistent data handle,

called a "moniker," which is a reference to the link source. A moniker is a data object that contains information to identify the linked data and provides methods through which an application program can bind to the linked data.

The binding process returns an instance of an interface through which the linked data can be accessed. In some cases, the moniker may link data that is itself embedded data within another compound document. For example, a moniker may link to a range of cells within a spreadsheet table that is contained in a word processing document. A moniker may link to data at any level within a compound document. During the binding process, several applications may be invoked to locate the linked data. For example, to link to the range of cells within a spreadsheet that is within a word processing document, the word processing program may be invoked to locate embedded spreadsheet data and the spreadsheet program may be invoked to bind to the range of cells.

The present invention defines an abstract class (an interface) through which a moniker is accessed. Typically, each application program that provides data which can be linked into a compound document supports an implementation of the moniker interface.

In a preferred embodiment of the present invention, an application program that creates a compound document controls the manipulation of linked or embedded data generated by another application. In object-oriented parlance, this data is referred to as an object. (The reference Budd, T., "An Introduction to Object-Oriented Programming," Addison-Wesley Publishing Co., Inc., 1991, provides an introduction to object-oriented concepts and terminology.) An object that is either linked or embedded into a compound document is "contained" within the document. Also, a compound document is referred to as a "container" object and the objects contained within a compound document are referred to as "containee" objects. Referring to Figures 1 and 2, the scheduling data 102 and

budgeting data 103 are containee objects and the compound document 101 is a container object. The user can indicate to the word processor that the user wants to edit a containee object, such as the budgeting data 103. When the user indicates that the budgeting data 103 is to be edited, the word processing program determines which application should be used to edit the budgeting data (e.g., the spreadsheet program) and launches (starts up) that application. The user can then manipulate the budgeting data using the launched application, and changes are reflected in the compound document. The same procedure is used whether the budgeting data is stored as an embedded or linked object.

Figure 3 is a block diagram showing a sample compound document. The weekly project report 301 is the same compound document of Figure 1. The executive summary report 303 contains a budgeting chart 305 that is linked to the weekly project report 301. The weekly project report 301 contains an embedded spreadsheet 302. The embedded spreadsheet 302 was created by the spreadsheet program 204 in Figure 2. The data for this spreadsheet, the budget for the project, is stored within the storage of the weekly project report 301 because it is an embedded object. The executive summary document 303 is a compound document which contains native text 304 and a contained object, the budget chart 305. The budget chart 305 is linked to the data contained within the spreadsheet 302 which is embedded in the compound document 301.

In a preferred embodiment, application programs ("applications") cooperate using object linking and embedding facilities to create and manipulate compound documents. An application that creates a compound document is referred to as a client application, and an application that creates and manipulates containee objects is referred to as a server application. An application can behave as both a client or a server. Referring to Figure

2, the project management program 201 and the spreadsheet program 204 are server applications, and the word processing program 206 is a client application. A client application is responsible for selection of the various objects within the container object and for invoking the proper server application to manipulate the selected containee object. A server application is responsible for manipulating the contents of the containee objects.

In a preferred embodiment, applications are provided with an implementation-independent Application Programming Interface (API) that provides the object linking and embedding functionality. The API is a set of functions that are invoked by client and server applications. These functions manage, among other things, the setup and initialization necessary for client applications to send and receive messages and data to and from server applications. The API provides functions to invoke the correct server application to act upon a particular containee object and to manipulate containee objects.

In addition, the object linking and embedding API defines "interfaces" through which client applications can communicate with their contained objects. An interface is a set of methods which abide by certain input, output, and behavior rules. If a contained object supports a particular interface, the client application can invoke the methods of that interface to effect the defined behavior. In a preferred embodiment, the client application is not allowed direct access to the object data; it must manipulate the object using the supported interfaces. A client application is bound to a contained object through a pointer to an interface. The client application accesses the object by invoking the methods of the interface. To access the object data, the methods may send messages to the server application requesting the specified access. In a preferred embodiment, messages are sent between clients and servers using interprocess

communications mechanisms provided by the underlying operating system.

An example will help illustrate the relationship between a client process and a server process. Referring again to Figure 1, if a user wants to edit the budgeting data 103 of the compound document 101, then the following sequence of events occurs. First, the user starts up the word processor program, which is dynamically linked to the object linking and embedding API. Second, the user opens the compound document for editing. Third, the user selects the budgeting data, which is a containee object, and indicates that the selected object is to be edited. Fourth, the client application invokes a client API routine for performing an action on an object passing the routine a handle (which uniquely identifies the selected object) to the object and an indicator that the action is edit. Fifth, the client API routine determines that the spreadsheet program provides the actions for the budgeting data. Sixth, the client API code starts up the spreadsheet program as a server process, if it is not already started. Seventh, the word processor application sends a message to the spreadsheet program that it should edit the budgeting data. Eighth, the server API code receives the request to edit and invokes a routine in the spreadsheet program for editing the data. When editing is complete, the spreadsheet routine returns to the server API code. The server API code sends a message to the word processor application to indicate that editing is complete. The client API code receives the message and returns from its invocation. Upon return from the invocation, the word processor application knows that the editing is complete.

In addition to the client and server APIs, the object linking and embedding facilities of the present invention provide information to client and server applications through a persistent global "registry." This registry is a database of information such as (1) for each

type of object, the server application that implements the object type, (2) the actions that the each server application provides to client applications, (3) where the executable files for each server application are located, and (4) whether each server application has an associated object handler.

In object-oriented parlance, an interface is an "abstract class": a class with a definition of its data and methods, but with no implementation for those methods. It is the responsibility of the server application for the object "class" to provide the actual code for the methods available to manipulate the object.

Figure 4 is a block diagram of a sample instance of an object. In a preferred embodiment, the layout of the instance conforms to the model defined in U.S. Patent No. 5,297,284, entitled "Method and System for Implementing Virtual Functions and Virtual Base Classes and Setting a This Pointer for an Object-Oriented Programming Language." The instance contains object data structure 401 and interface data structure 413 for each supported interface. The object data structure 401 contains pointers 402 to the interface data structures 413 and may contain private data of the instance. The private data of this sample instance includes a class identifier 403, handle 404 to the storage for the object, and data 405 for tracking the state of the object. The class identifier is used to access the appropriate server application for the object. It is similar to a data structure "type" used in programming languages. The interfaces can determine the server for the object by using the CLASS_ID as an index into the persistent global registry. As shown in Figure 4, each interface data structure 413 contains a private data structure 406 and a virtual function table 408. The private data structure 406 contains a pointer 407 to the virtual function table 408. The virtual function table

408 contains pointers to the code that implements the methods of the interface.

TABLE 1

```
# define interface class
interface intf {public:
     virtual RETCODE pfnc1 (arg1, arg2) = 0;
     virtual RETCODE pfnc2 (arg1, arg2) = 0;
     virtual RETCODE pfnc3 ( ) = 0;
                    . . .
};
```

Table 1 represents the definition for the interface for the first entry $pintf_1$ in the object data structure 401. In Table 1, the word "interface" is defined to mean a C++ class. The definition shows three methods with their arguments. The "=0" at the end of each argument list indicates that the method has no code implementation. In the C++ programming language, these functions are termed "pure virtual functions". A class with all pure virtual functions is referred to as an abstract class.

Figure 5 is a block diagram showing public view of an object. A public view of an object is the various interfaces that the object supports 502-506. Each interface provides methods through which client applications can access the object. Each object supports an IUnknown interface 502. Applications use the IUnknown interface 502 to determine which other interfaces the object supports. The implementation of IUnknown interface 502 for a particular object knows what other interfaces it supports and returns to the invoking application pointers to those interfaces. In a preferred embodiment, the method IUnknown::QueryInterface is used for this purpose. Interfaces 503 through 506 are examples of typical interfaces that can be supported by an object. For example, the ICreate interface 503 provides methods for

creating a new instance of the object. The IContainer interface 504 provides methods for listing the containee objects that are contained within the object. The IDataClientSite interface 505 provides methods to be used by the servers communicating with the container object. The IDataObject interface 506 provides methods for manipulating the object data.

Figure 6 is a block diagram showing a typical instance of the IMoniker interface. Appendix A contains a detailed description of the methods of the IMoniker interface. The instance data structure includes object data structure 601 and virtual function table 603. The object data structure 601 contains a pointer 602 to the virtual function table 603 and private instance data. Virtual function table 603 contains pointers to the code that implements the method of the IMoniker interface. Table 2 is the class definition for the IMoniker interface.

TABLE 2

```
class IMoniker
{
virtual    SCODE    BindToObject(pbc, iidToLeft, pvToLeft, pmkToLeft, IidResult, ppvResult)
= 0;
virtual    SCODE    IIdBindToObject(pidd) = 0;
virtual    SCODE    BindToStorage(pbc, pmkToLeft, iid, ppvObj) = 0;
virtual    SCODE    Reduce (pbc, dwReduceHowFar, ppmkToLeft, ppmkReduced) = 0;
virtual    SCODE    ComposeWith(pmkRight, fOnlyifNotGeneric, ppmkComposite)
virtual    SCODE    Enum(fForward, ppenmMoniker) = 0;
virtual    SCODE    IsGeneric Composite() = 0;
virtual    SCODE    IsEqual(fIdnetical, pmkOtherMoniker) = 0;
virtual    SCODE    Hash(fIdentical, pdwHash) = 0;
virtual    SCODE    GetTimeOfLastChange(pbc, pmkToLeft, pfiletime) = 0;
virtual    SCODE    Inverse(ppmk) = 0;
virtual    SCODE    CommonPrefixWith(pmkOther, ppmkPrefix) = 0;
virtual    SCODE    RelativePathTo(pmkOther, ppmkRelPath);
virtual    SCODE    GetUserName(pbc, pmkToLeft, LpLpszUserName) = 0;
```

```
virtual    SCODE      ParseUserName(pbc, pmkToLeft, lpszUserName, pchEaten, pppmkOut) =
0;
}
```

In addition to these methods, an IMonkier implementation typically supports the functions MkParseUserName, MkCreateIBindContext, MkCreateGenericComposite, MkCreateFileMoniker, MkCreateItemMoniker, and MkCreateAntiMoniker, which are defined below.

Figure 22 is a diagram illustrating a linked object. In this example, the compound document 2201 is a compilation of various project reports, such as the report stored in compound document 101. Compound document 2201 contains a link to the compound document 2202. The link indicates that the compound document stored in the file "VAC1.DOC" is logically within the compound document 2201. In a preferred embodiment, the link is stored as a moniker. A moniker is an instance of a class (a moniker object) that provides methods to access the source of the link. A moniker object encapsulates data that identifies the link source and provides methods through which an application program can access the link source. For example, when the link is created from compound document 2201 to compound document 2202, the word processor invokes a function to instantiate a moniker object that points to the "VAC1.DOC" file. The function returns a pointer to the new moniker object. The word processor then stores the moniker object within the compound document 2201. The word processor accesses the link source through the methods of the moniker object. For example, the moniker class supports a "bind" method. The bind method determines where the link source is located and returns a pointer to an object which represents the source. In this example, the bind method of the moniker object would determine that the source is stored in file "VAC1.DOC," instantiate an object to represent a file, and return a

pointer to the instantiated object. The word processor could then invoke the methods of the returned file object to access the source data.

Figure 7 is a block diagram showing the moniker that is stored in the executive summary compound document 303 in Figure 3. This moniker represents the linked chart object 305. Because the link points to data embedded within a compound document, the moniker is represented as a composite of a link needed to identify the compound document and a link needed to identify the data within the compound document. The link is referred to as a composite moniker. A composite moniker contains component moniker ordered in a left to right manner in an embedding hierarchy. The composite moniker 701 comprises three component monikers, 702, 703, and 704. Component moniker 702 references the compound document 301. Component moniker 702 contains a CLASS_ID (FileMoniker), a user readable name "C:\VAC1.DOC" of the compound document 301. Component moniker 703 references the embedded object within compound document 301. The word processing application understands component moniker 703 and is used to locate the embedded spreadsheet. Component moniker 703 contains CLASS_ID (WItemMoniker) and a user readable name of the embedded spreadsheet. The CLASS_ID "WItemMoniker" indicates that the word processor recognizes this component moniker. Component moniker 704 references the range of data within the embedded spreadsheet object 302. Component moniker 704 contains CLASS_ID (SItemMoniker) and a user readable name for the range. The CLASS_ID "SItemMoniker" indicates that the spreadsheet program recognizes this component moniker. The composite moniker object data structure includes data structure 801 and virtual function table 802. Data structure 801 contains the CLASS_ID 803 and a pointer 804 to the linked list 805. The CLASS_ID, in this example, indicates the moniker of type ID_GenCompositeMoniker.

Figure 8 is a block diagram showing a typical composite moniker object data structure. The linked list 805 contains one entry 806 for each component moniker. Each entry contains a pointer to the component moniker 809-811.

Figure 9 is a flow diagram of an implementation of the generic composite moniker method ComposeWith. The method GenCompositeMoniker::ComposeWith creates a new composite moniker with the specified moniker on the left and the designated moniker as the right portion of the new composite moniker. This method returns a pointer to the new composite moniker. In step 901, the method calls the IsGenericComposite? method of the designated moniker to determine whether it is a generic composite moniker. If it is not, the method continues at step 904, otherwise it continues at step 902. In step 902, the method invokes the FindFirstElement method of the designated moniker, which returns a pointer to the first component moniker and pointer to the rest of the component moniker. In step 903, the methods extracts the CLASS_ID of the first component moniker. In step 904, the function extracts the CLASS_ID of the designated moniker. When steps 903 and 904 are complete, the method has obtained the CLASS_ID of the moniker it will use as the right portion of the new composite moniker. In step 907, the method invokes the MkCreateGenericComposite function to create a new generic composite moniker specified moniker on the left and the designated moniker on the right, and then the method returns with a pointer to the new composite moniker.

Figure 10 is a flow diagram for the function MkCreateGenericComposite. This function takes two monikers, a left and a right, creates a new composite moniker comprising the left and right monikers and returns a pointer to the new moniker. In step 1001, the function

calls the pmkFirst::IsGenericComposite? method to determine whether the left moniker is a generic composite moniker. If it is, the function continues at step 1004, otherwise, it continues at step 1002. In steps 1002 and 1003, the function allocates a new generic composite moniker and stores a copy of the left moniker as its first element. In step 1002, the function allocates the necessary data structures and allocates a new generic composite moniker node in the linked list and stores this value in the variable previous. In step 1003, the function inserts the left-hand moniker passed in (pmkFirst) as the first element in the new composite moniker pointed to by pcm. Then the method continues at step 1008. In step 1004 through 1007, the function loops to create a new composite moniker that contains the component monikers of the original left moniker. The loop traverses the elements of the linked list of the original left moniker and inserts a copy of the component monikers into the new composite moniker. In step 1004, the function initializes several variables in order to search for the last element of the left moniker. To do this, it sets the variable previous to point to the first element of the linked list, the variable current to point to the second element of the linked list, and pcm to the first moniker pointed by the first element of the linked list. In step 1005, the function determines whether the variable current is NULL. If it is, then variable previous points to the last element in the left composite moniker which indicates that all the component moniker have been invalid into the new composite moniker and the function continues its step 1008, otherwise, the function continues at step 1006. In step 1006, the function invokes the ComposeWith method of the moniker pointed to by the variable pcm passing it the moniker pointed to by the variable current. In step 1007, the function sets the variable pcm to point to this new composite moniker and updates the variables previous and current to point to the next two elements in

the linked list and loops to step 1005 to determine whether the variable previous points to the end of the linked list. In step 1008, the function calls pmkRestIsGenericComposite? method to determine whether the right moniker is a generic composite moniker. If it is not, the function continues its step 1010, else it continues at step 1009. In step 1009, the function appends the linked list of the right moniker to the new left moniker. The function then continues in step 1012. In steps 1010 and 1011, the function appends a new element pointing to the right moniker to the new left moniker. In step 1012, the function sets the return value ppmkComposite to the pcm, the newly composed moniker, and returns.

Figure 11 is a flow diagram of a FindLastElt method of a typical implementation. This method creates two monikers. The first moniker corresponds to the last component moniker in the specified moniker. The second moniker corresponds to the remainder of the specified moniker. In step 1101, the method sets the variable current to point to second element in the linked list, the variable previous to the first element in the linked list, and the variable rest to point to the first component moniker. Steps 1102 through 1105 traverse the linked list of the specified moniker creating the second moniker. In step 1102, if the variable current points to the last element, the method continues at step 1106, else the method continues at step 1103. In step 1106, the method sets the parameter LastElt to point to the moniker pointed to by variable current, and returns.

Figure 12 is a flow diagram for the FindFirstElt method of a typical implementation. This method is analogous to the FindLastElt method but returns a pointer to the first component moniker and a pointer to a new composite moniker comprising the remaining component monikers.

Figure 13 is a flow diagram of the BindToObject method for a typical implementation. The method BindToObject recursively invokes either BindToObject (or BindToStorage) to obtain an object (or storage) that can be used to locate and bind to the object indicated by the specified moniker. In general, the BindToObject method returns the object or storage indicated by the specified moniker using the method of its prefix component moniker. This method invokes the BindToObject method of the last component moniker, which in turn invokes the BindToObject methods of its prefixmoniker in a right to left manner. The BindToObject method takes six parameters: a pointertoabind context, the interface identifier of the third parameter, the pointer to an interface of the object corresponding to the prefix moniker, the prefix moniker, the interface to which the caller wishes to bind, and an output parameter corresponding to the instantiated interface. In step 1301, the method invokes the FindLastElt method of the specified moniker to find its last component moniker. In step 1302, the method invokes the IidBindToObject method of the last component moniker. In step 1303, if this method returns ID_NULL, then the BindToObject method of the last component moniker needs to be invoked to determine the interface that is required for binding and the method continues at step 1304, else the method continues at step 1305. In step 1304, the BindToObject method invokes the BindToObject method of the last component moniker, passing it the first part of the composite moniker as its prefix moniker and returns. In step 1305, if the interface returned by the IidBindToObject method of the last component moniker is equal to ID_STORAGE, then the method continues at step 1306, else it continues its step 1307. In step 1306, the method invokes the BindToStorage method of the prefix

moniker (variable rest). When it returns, variable pv points to a bound instance of the storage object corresponding to the prefix moniker and continues at step 1308. In step 1307, the method instead invokes the BindToObject method on the prefix moniker. In this case, it passes the interface requested by the last component moniker and the instantiated object corresponding to that interface is returned in the variable pv. In step 1308, the instantiated object pointed to by variable pv is registered in the bind context passed in. The bind context is a linked list of currently instantiated objects correspondingly bound by previous invocations of the BindToObject method. The bind context is used by the BindToObject methods determine whether the object referred to by the specified moniker is already instantiated. In step 1309, the method invokes the BindToObject method for the last component moniker with the interface requested by the last component moniker, the instantiated interface corresponding to the moniker prefix, and the original interface requested in the original BindToObject call. The method then returns.

Figure 14 is a flow diagram of the BindToObject method for a typical implementation of a file moniker. This method determines whether an object corresponding to the specified file moniker exists as indicated by a Running Object Table and returns a pointer to the object. The Running Object Table contains pointer to all current instantiated object. Otherwise, this method locates the appropriate server and instantiates an interface to the file. The FileMoniker::BindToObject method takes the same six parameters listed in the generic composite moniker BindToObject method. In steps 1401 through 1404, the method uses the Running Object Table to an object corresponding to determine whether the specified file moniker exists. In step 1401, the method determines whether the specified moniker is listed in the Running Object Table. If it is, the method continues at step

1402, otherwise it continues at step 1405. In step 1402, the method determines whether the requested interface matches that in the Running Object Table. If it does, then in step 1403 the method returns a pointer to the interface. Otherwise, in step 1404, the method obtains the requested interface the by calling the QueryInterface method of the object found in the Running Object Table and returns. In step 1405, because the specified moniker was not found in the Running Object Table, this method finds the server corresponding to the specified moniker (filename). In step 1406, the method calls the server to create an instance of its class. In step 1407, the method calls the QueryInterface method of this instance to obtain the interface for binding. In step 1408, the method calls the Load method of the binding interface and returns.

Figure 15 is a flow diagram of the BindToObject method for a typical implementation of an item moniker. In the above example, WItemMoniker and SItemMoniker both have similar implementations. This method uses the interface object of the prefix moniker to invoke its GetObject method. The GetObject method locates and binds to an instance of the object indicated by the ItemMoniker. This method takes the same six parameters passed to the BindToObject method of the generic composite moniker. In step 1501, the method determines whether an object was passed for binding. If it was, the method continues its step 1502, otherwise it continues at step 1503. In step 1502, the method invokes the GetObject method of the interface passed in to instantiate the object indicated by the item moniker and then return. In step 1503, if the prefix moniker is not NULL, the method continues at step 1504, otherwise it returns. In step 1504, the method invokes the BindToObject method of the prefix moniker, requesting the IContainer interface. In step 1505, the method invokes the GetObject method of the requested IContainer interface and returns.

Figure 16 is a flow diagram of the BindToStorage method for a typical implementation of WItemMoniker. This method invokes the BindToStorage method of its prefix, and then, once it has obtained the storage handle for the prefix, it gets the storage associated with the specified moniker. In step 1601, the method determines whether the prefix passed in is NULL, and if it is, the method returns, otherwise the method continues its step 1602. In step 1602, the method invokes the BindToStorage method of the prefix moniker and passes it the requested interface. In step 1603, this method invokes the GetStorage method of the storage interface returned in step 1602 passing it the unique identifier for the object specified in the current WItemMoniker. The method then returns.

Figure 17 is a overview flow diagram for the Reduce method for the GenericCompositeMoniker interface. This routine attempts to reduce each component moniker by calling the reduce method of each component moniker and composes a new GenericCompositeMoniker with the reduced components. The method takes three parameters: a pointer to a bind context, a prefix moniker, and an output parameter for the reduced moniker. In step 1701, the method finds the first element in the linked list of component monikers comprising the specified moniker. In steps 1702 through 1709, the method iteratively invokes the Reduce method of each component moniker and produces a new composite moniker with reduced components. In step 1702, if all the elements of the linked list have been processed, the method continues at step 1703, otherwise it continues at step 1704. In step 1703, the method sets the output parameter to the new reduced composite moniker and returns. In step 1704, the method invokes the Reduce method of the moniker component pointed to by the variable M_Element, passing along the bind

context and a pointer to the prefix moniker. The prefix moniker corresponds to the new composed moniker produced so far. In step 1705, if the method determines that the variable pointing to prefix moniker is NULL, it continues at step 1706, otherwise it continues at step 1707. In step 1706, if the variable pointing to the prefix moniker points to NULL (which happens on the first time through the loop), it is set to the reduced component moniker returned in step 1704 and the method continues at step 1709. In step 1707, if the variable pointing to the prefix moniker is not NULL, the method invokes the ComposeWith method of the prefix moniker to compose the prefix with the new reduced moniker returned in step 1704. In step 1708, the variable pointing to the prefix moniker is reset to the new composed result, which contains all of the moniker reductions that have occurred. In step 1709, the variable M_Element is advanced to the next element in the linked list of component monikers and the method returns to the top of the loop at step 1702.

Figure 18 is a flow diagram of the Reduce method for a typical implementation of a FileMoniker. The Reduce method looks at the prefix moniker passed in and determines whether last component of the prefix moniker is another file moniker. If it is another File Moniker, this method returns a new composed FileMoniker that is the combination of the two FileMonikers into one. It also removes the file moniker it consumed from the prefix passed in. This function takes three parameters: a pointer to a bind context, a pointer to the prefix moniker, and an output parameter where the newly reduced moniker is returned. In step 1801, the function examines the prefix moniker passed in to see if it is NULL. If it is, the function returns, otherwise it continues at step 1802. In step 1802, the function looks at the prefix moniker to see if it is a generic composite moniker. If it is, the function continues at step 1804, else it continues at step 1803. In step 1803, the function sets

the last element to the prefix moniker because the prefix is a simple moniker and continues at step 1805. In step 1804, the function invokes the FindLastElt method of the prefix moniker in order to obtain the last element of the prefix composite moniker. In step 1805, the CLASS ID of the last element is checked to see if it is a file moniker. If it is not, the function continues at step 1806, otherwise it continues at step 1807. In step 1806, the function sets the output parameter to itself, because no reduction has occurred, and returns. In step 1807, the function appends the filename associated with this moniker to the filename of the moniker pointed to by the last element. Next, step 1808, the function invokes its own create method with this new filename string. In step 1809, it sets the prefix equal to what was the prefix minus the last element, or NULL if the prefix was not a composite moniker. In step 1810, the function sets the output parameter corresponding to the newly reduced moniker to the moniker created in step 1808 and the function returns.

Figure 19 shows an overview flow diagram of the Reduce method for a typical implementation of the word processing item moniker WItemMoniker. This routine returns a special kind of moniker, a storage moniker, which can be used by the bind-to-object method to invoke the server for an embedded object directly, thereby bypassing invoking the word processing application. It does this by looking in a special object table contained within the word processing document to find the appropriate storage pointer for the embedded object. This storage pointer is then composed in a storage moniker, and when its bind-to-object method is invoked, it can figure out which server to call. This routine takes three parameters: a pointer-to-a-bind context, a prefix moniker, and an output parameter that will contain the newly reduced moniker. In step 1901, the method looks to see if the prefix is Null, and if it is returns,

otherwise, the method continues at step 1902. In step 1902, the method looks to see if the prefix is a generic composite moniker, and if it is, continues at step 1903, else it continues at step 1904. In step 1903, the function invokes the prefix moniker's FindLastElt method to obtain a pointer-to-last moniker in the linked list. It then continues at step 1905. In step 1904, the method sets the last element pointer to the prefix because it is not a composite moniker. In step 1905, the method looks at the CLASS_ID of the last element to see if it is file moniker. If it is not, the function continues at step 1906, else it continues at step 1907. In step 1006, the function simply returns itself in the output parameter, because no reduction was possible. In step 1907, the method looks at the file extension or however it knows to determine whether the file belongs to the word processing application. If it determines the file is of an unknown type, the function continues at step 1906, else the function continues at step 1908. In step 1908, the method invokes the GetUser method of the moniker pointed to by the last element pointer in order to get the file name of the last element with prefix. In step 1909, it looks in the object table at a known location in the file name it obtained to find the entry corresponding to the current moniker. The elements in the table are in the form of a three tuple (WItemMoniker, IStorage, UserName). From this table, the method gets the storage pointer and the user name for the new moniker it will create. Next, in step 1910, it appends the username of the filename of the last element of the prefix to this new name it has obtained. In step 1911, the method creates a new IStorage moniker with the storage pointer and the new name. In step 1912, the prefix is set to the remainder of what was the prefix minus the last element. Finally, in step 1913, the output parameter is set to the new IStorage moniker created and the function returns.

Figure 20 shows an overview flow diagram for the function MkParseUserName. This function takes a string name and returns its corresponding moniker. The moniker returned may be a composite moniker. In step 2001, the function parses the string to find a legitimate filename. In step 2002, the function creates a file moniker for this filename and saves it in the variable pfm. In step 2003, the function sets the string remaining to the initial string passed in minus the characters that were consumed. Finally, in step 2004, the function invokes the ParseUserName method for the file moniker pointed to by pfm and passes it the string remaining and the address of the return moniker. Once the file moniker ParseUserName method has returned, the rest of the string will have been parsed. The function then returns.

Figure 21 shows an overview flow diagram for the method ParseUserName for a typical implementation of a file moniker. Because it does not know the kinds of objects it can contain, the method must decide whether it needs to bind to the object indicated by the filename in order to parse the next part of the string. This method takes four parameters: a pointer to a BindContext, the pointer to the prefix moniker, the string name remaining to be parsed, and a pointer to the resulting moniker. In step 2101, the method decides whether it can understand the syntax of the objects that are contained within this file. If it can, the method continues at step 2102, otherwise it continues at step 2103. In step 2102, the method calls its internal parse string routine and creates a moniker for the portion of the string it was able to parse. It then continues at step 2105. In 2103, the method calls BindToObject on itself, passing it the prefix that was passed to the ParseUserName method. Next, in step 2104, this method calls the ParseUserName method of the object that was bound to passing it the prefix and the user string remaining to be parsed. Finally, in step

2105, the ComposeWith method of the prefix is called to add the newly produced moniker. The function returns.

Although the present invention has been described in terms of a preferred embodiment, it is not intended that the invention be limited to his embodiment. Modifications within the spirit of the invention will be apparent to those skilled in the art. The scope of the present invention is defined by the claims which follow.

CLAIMS

1. A method in a computer system for linking an object within a first compound document to data that resides in a second document, the method comprising the steps of:

(a) instantiating a moniker object, the moniker object for storing information specifying the data in the second document and for providing a function for binding to the data in the second document;

(b) storing the moniker object in the first document; and

(c) invoking the binding function of the stored moniker object to bind to the data in the second document.

2. The method of claim 1 wherein the storing of the moniker object stores a reference to the moniker object.

3. The method of claim 1 wherein the function instantiates a source object for the data and returns a pointer to the instantiated source object.

4. The method of claim 1 wherein the instantiated moniker object is a composite moniker object.

5. The method of claim 4 including storing in the composite moniker object a reference to a first component moniker object and a reference to a second component moniker object wherein the second moniker object is composed with the first component moniker object to refer to an item within the data.

6. The method of claim 5 wherein the binding function invokes the binding function of the first component moniker object and the binding function of the first component moniker object invokes the binding function of the second component moniker object.

7. The method of claim 1 wherein the storing of the moniker object includes storing an identifier of a class of the moniker object along with information for locating the data that resided in the second compound document.

8. The method of claim 1 wherein the storing of the moniker object includes storing a filename of the second document.

9. The method of claim 1 wherein the storing of the moniker object includes storing a filename of the second document along with an indication of a portion of the second document.

10. The method of claim 1 wherein the binding function launches a server to provide access to the data.

11. The method of claim 1 wherein the moniker object provides a function for determining other interfaces supported by the moniker object.

12. The method of claim 1 wherein the moniker object provides a function for composing the moniker object with another moniker object to form a composite moniker object.

13. The method of claim 1 wherein the moniker object accesses the data without launching a server for processing the data.

14. The method of claim 1 wherein the instantiating of the moniker object includes parsing a string name for the data.

15. A method in a computer system of creating an object that refers to a source object, the method comprising the computer-implemented steps of:

(a) creating an instance of an identifier object of an identifier class, the identifier class having a binding member function that, when invoked, binds to the source object by locating and accessing the source object;

(b) storing a reference to the source object in the created identifier object as naming information that is used by the binding member function to bind to the source object; and

(c) storing a reference to the created identifier object having the stored reference to the source object that is used when binding to the source object.

16. The method of claim 15 wherein the binding member function of the created identifier object, when invoked, binds by further instantiating the source object referred to by the naming information of the created identifier object and returning a pointer to the instantiated source object.

17. The method of claim 15, the identifier object having a reducing member function for creating a reduced identifier object that is a canonical form of the identifier object and that identifies the source object.

18. The method of claim 15, the computer system having client code, the source object associated with server code, wherein the step of creating the instance of the identifier object is performed by executing the server code, the method further comprising the steps of:

(a) terminating the execution of the server code after creating the instance of the identifier object; and

(b) after terminating the execution of the server code, locating and accessing the source object by executing the binding member function of the created identifier object, wherein the binding member function launches the server code to locate and access the source object referred to by the naming information of the created identifier object.

19. A method in a computer system for composing a first identifier object with a second identifier object to create a composite identifier object that refers to a source object, the first identifier object having an internal state that refers to a container object and having a first class identifier that identifies binding code that, when invoked, locates and connects to the container object referred to by the internal state, the second identifier having an internal state that refers to an item and having a second class identifier that identifies binding code that, when invoked, locates and connects to the item referred to by the internal state, the method comprising the computer-implemented steps of:

(a) creating an instance of a composite identifier object, the created composite identifier object having an internal state and having a composite class identifier that identifies binding code that, when invoked, locates and connects to the source object referred to by the internal state of the created composite identifier object;

(b) storing a reference to the first identifier object in the internal state of the created composite identifier object; and

(c) storing a reference to the second identifier object in the internal state of the created composite identifier object, wherein the second identifier object is composed with the first identifier object to refer to the item within the container object as the source object.

20. A method for binding to a source object in accordance with the method of claim 19, further comprising the steps of:

(a) when the binding code identified by the composite class identifier is invoked, invoking the binding code identified by the second class identifier of the second identifier object passing the stored reference to the first identifier object; and

(b) when the binding code identified by the second class identifier of the second identifier object is invoked,

(i) invoking the binding code of the first identifier object to bind to the container object referred to by the internal state of the first identifier object;

(ii) receiving a reference to the bound container object referred to by the first identifier object; and

(iii) using the received reference to the bound container object to locate and connect to the item referred to by the internal state of the second identifier object, thereby binding to the source object referred to by the composite identifier object.

21. The method of claim 19, the binding code identified by the composite class identifier of the composite identifier object having a bind context that contains indicators to instantiated objects created in the process of binding to the source object, wherein the bind context is passed, when the binding code identified by the class identifier of the second identifier object is invoked and when the binding code identified by the class identifier of the first identifier object is invoked, to prevent unnecessary creation of already existing objects.

22. The method of claim 19, the internal state of the composite identifier object being structured in a composition sequence from left to right such that the reference to the first identifier object is conceptually to the left of the reference to the second identifier object in the composition sequence and such that the reference to the second identifier object is conceptually at the rightmost end of the composition sequence.

23. A computer system for creating an object that refers and binds to source data comprising:

(a) means for instantiating an identifier object having a class identifier that identifies binding code; and

(b) means for storing in the instantiated identifier object a reference to the source data as identifying data, wherein the binding code, when invoked, locates and connects to the source data referred to by the stored identifying data; and

(c) means for storing in the instantiated identifier object a class identifier that identifies the binding code.

24. The system of claim 23, further comprising means for storing the instantiated identifier object in a persistent medium.

25. A method in a computer system for binding to source data, the method comprising the computer-implemented steps of:

(a) creating an instance of an identifier object, the created identifier object having naming information that identifies the source data and having a class identifier that identifies binding code that, when invoked, binds by locating and connecting to the source data identified by the naming information;

(b) invoking the binding code identified by the class identifier of the created identifier object to bind to the source data; and

(c) when the binding code is invoked, returning an indicator of the bound source data.

26. The method of claim 25 wherein the source data is stored in a file, the source data having a data type that indicates associated server code that, when executed, manages the source data, wherein the naming information that identifies the source data is a file name, the file name indicating the data type of the source data stored within the file, and wherein the binding code identified by the class identifier of the identifier object, when executed, invokes the server code associated with the data type indicated by the file name stored as the naming information, requests the invoked server code to load the source data from the file referred to by the file name, and returns a reference to the loaded data.

27. The method of claim 25, further comprising the step of storing the created identifier object in a persistent medium.

28. A method in a computer system of generating an object that refers to source data, the computer system having a destination object, the method comprising the computer-implemented steps of:

(a) creating an instance of a moniker object, the created moniker object having a moniker class identifier that identifies binding code that, when executed, binds to the source data;

(b) storing a reference to the source data in the created moniker object as naming information;

(c) storing the class identifier in the created moniker object; and

(d) storing in the destination object a reference to the created moniker object having the stored reference to the source data.

29. The method of claim 28, further comprising the step of accessing the source data from the destination object by retrieving the reference to the created moniker object that is stored in the destination object, and invoking the binding code identified by the moniker class identifier of the created moniker object referred to by the retrieved reference to resolve the stored reference to the source data into locatable data and to locate and connect to the locatable data.

30. The method of claim 29, the source data having associated server code that, when executed, manages the source data, wherein the step of accessing the source data by invoking the binding code identified by the moniker class identifier of the created moniker object further comprises the substeps of:

(a) locating the server code associated with the source data referred to by the naming information of the created moniker object referred to by the retrieved reference;

(b) invoking the located server code;

(c) requesting the invoked server code to connect to the source data referred to by the naming information; and

(d) when the server code is invoked,

(i) instantiating a source object;

(ii) loading the source data referred to by the naming information into the instantiated source object; and

(iii) returning an indication of the instantiated source object having the loaded source data.

31. The method of claim 29, the computer system having a computer memory and having a source object that contains the source data, the source object having server code that, when executed, manages the source object, wherein the step of accessing the source data by invoking the binding code identified by the moniker class identifier of the created moniker object further comprises the substeps of:

(a) determining whether an instance of the source object already exists in the computer memory and whether the server code is already launched;

(b) when it is determined that the instance of the source object already exists and the server code is already launched, obtaining an indicator to the instance of the source object; and

(c) returning the indicator.

32. The method of claim 31 wherein the step of determining whether the instance of the source object already exists further comprises the step of determining whether an entry for the created moniker object exists in a running object table.

33. A method in a computer system of binding to source data having associated persistent storage for containing the source data, the method comprising the computer-implemented steps of:

(a) instantiating an identifier object, the instantiated identifier object having a class identifier that identifies binding code that, when executed, binds to the source data;

(b) storing a reference to the source data in the instantiated identifier object as naming information; and

(c) accessing the source data by invoking the binding code identified by the class identifier of the instantiated identifier object to obtain an indicator of the persistent storage associated with the source data.

34. A method in a computer system of generating a reduced reference to source data, the method comprising the computer-implemented steps of:

(a) creating an instance of a moniker object, the created moniker object having a moniker class identifier that identifies a reduce member function;

(b) storing a reference to the source data in the created moniker object as naming information; and

(c) reducing the stored reference to the source data to a canonical form by invoking the reduce member function, wherein the reduce member function creates a reduced moniker object, the created reduced moniker object having reduced naming information that refers, in the canonical form, to the source data referred to by the naming information of the created moniker object.

35. The method of claim 34 wherein the step of reducing the stored reference to the source data is repeated using the created reduced moniker object until no further reductions are possible.

36. The method of claim 34, further comprising the steps of:

(a) requesting a number of times to perform the step of reducing; and

(b) repeating, the requested number of times, the step of reducing the stored reference to the source data using the created reduced moniker object for each subsequent step of reducing.

37. The method of claim 34 wherein the stored reference to the source data contains a query that, when executed, identifies the source data and wherein the step of reducing the stored reference executes the query to create the reduced moniker object.

38. The method of claim 37, the reduced moniker object having a class identifier that identifies a binding member function that, when invoked, uses the reduced naming information to locate the source data, the moniker class identifier identifying a binding

member function to locate and connect to the source data referred to by the naming information of the created moniker object, wherein the binding member function of the created moniker object, in response to being invoked, invokes the reduce member function of the created moniker object to obtain the reduced moniker object and then invokes the binding member function identified by the class identifier of the reduced moniker object to locate and connect to the source data.

39. A computer system for binding to source data comprising:

(a) means for generating a moniker object having naming information that identifies source data and having a class identifier that identifies a binding member function that, when executed, binds to the source data identified by the naming information and returns an indicator of the bound source data; and

(b) means for invoking the binding member function of the moniker object.

40. A method in a computer system for binding to a source object, the computer system having a destination object, the method comprising the computer-implemented steps of:

(a) instantiating a first moniker object of a moniker class, the moniker class identified by a moniker class identifier and having binding code that, when invoked, connects to the source object;

(b) storing a source reference to the source object in the instantiated first moniker object as naming information, the source object associated with server code;

(c) storing in the destination object a moniker reference to the instantiated first moniker object having the stored source reference;

(d) storing the destination object in persistent storage including,

(i) accessing the moniker class identifier and the naming information referred to by the moniker reference;

(ii) storing in persistent storage the accessed moniker class identifier; and

(iii) storing in persistent storage the accessed naming information;

(e) when the destination object is accessed from persistent storage,

(i) retrieving the stored moniker class identifier and naming information from persistent storage;

(ii) instantiating a second moniker object of the moniker class identified by the retrieved moniker class identifier;

(iii) storing the retrieved naming information in the instantiated second moniker object; and

(iv) invoking the binding code of the moniker class;

(f) when the binding code is invoked,

(i) invoking the server code associated with the source object referred to by the naming information stored in the second moniker object; and

(ii) requesting the server code to connect to the source object referred to by the naming information stored in the second moniker object; and

(g) when the server code is invoked,

(i) instantiating the source object referred to by the naming information stored in the second moniker object;

(ii) loading data into the instantiated source object; and

(iii) returning to the invoked binding code an indication of the instantiated source object with the loaded data, wherein the loaded data of the instantiated source object is accessed using the returned indication.

41. A method in a computer system of composing a first moniker object with a second moniker object, the method comprising the computer-implemented steps of:

(a) creating an instance of the first moniker object, the created first moniker object having a moniker class identifier that identifies a compose function member that, when executed, creates a composite moniker object that refers to the composition of the first moniker object with an indicated moniker object;

(b) creating an instance of the second moniker object; and

(c) invoking the compose function member identified by the moniker class identifier of the first moniker object, using the created second moniker object as the indicated moniker object, to create an instance of a composite moniker object that

refers to the composition of the first moniker object with the second moniker object.

42. The method of claim 41, the first moniker object having naming information that refers to first data, the second moniker object having naming information that refers to second data, wherein the step of invoking the compose function member of the first moniker object combines the naming information of the second moniker object with the naming information of the first moniker object using knowledge of the structure and semantics of the second moniker object when creating the instance of the composite moniker object.

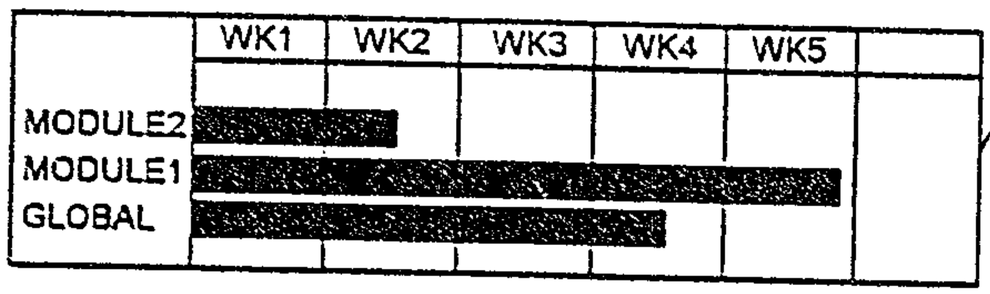
43. The method of claim 41, further comprising the step of storing the created composite moniker object.

101

VAC1.DOC

VAC1 PROJECT July 1, 1990

SCHEDULE:



102

TABLE: SCHEDULE

104

BUDGET:

ITEM	EST $	EXP TO DATE	DELTA	RUNNING
SUPPLIES	100.00	50.00	+50.00	50.00
COMPUTERS	4000.00	3895.00	+195.00	3945.00
MANUALS	500.00	500.00	0.00	4445.00
TOTAL	5600.00	4445.00	+245.00	4445.00

TABLE 2: VAC1 BUDGET

103

Figure 1

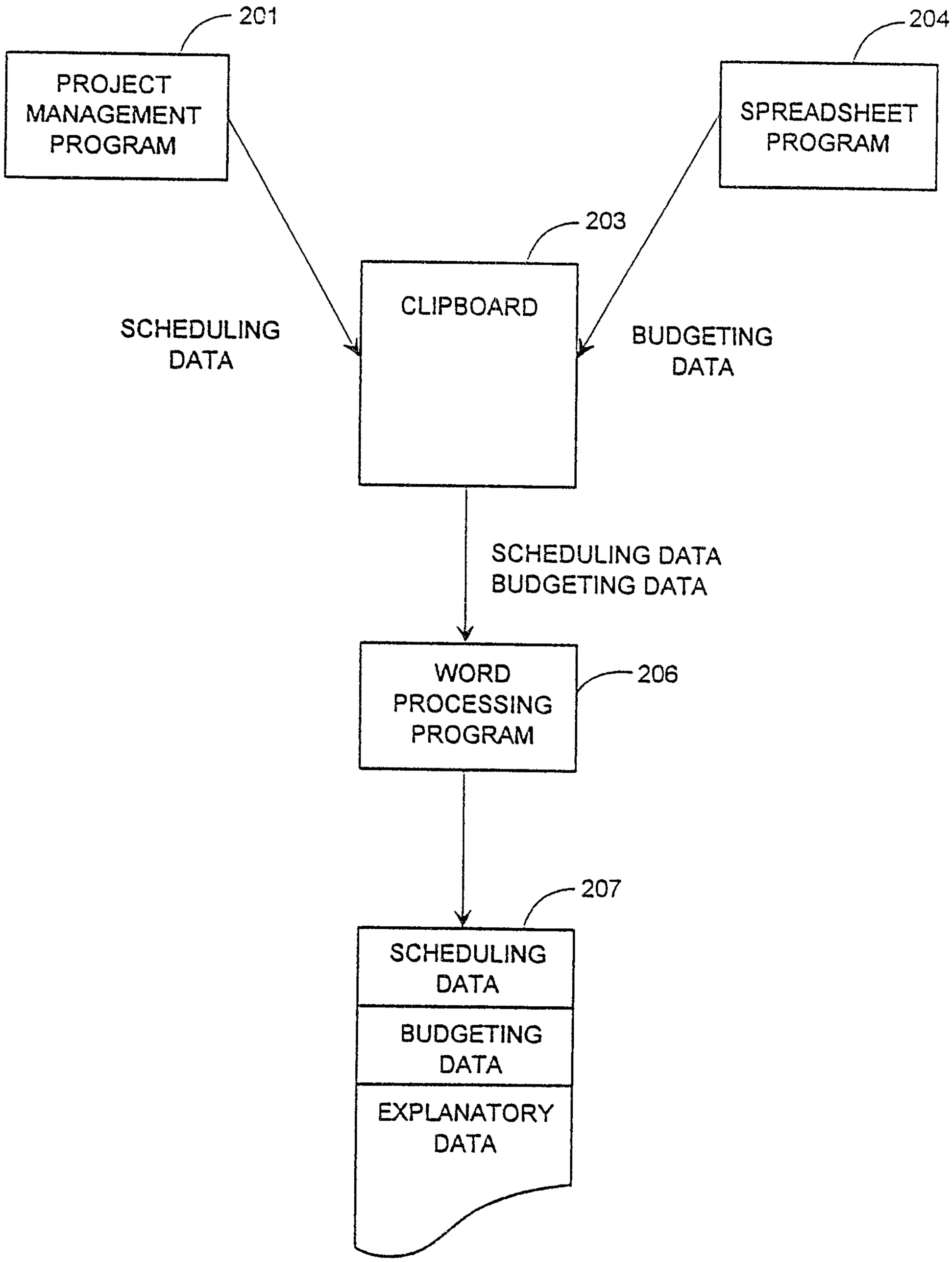

201
PROJECT MANAGEMENT PROGRAM
204
SPREADSHEET PROGRAM
203
CLIPBOARD
SCHEDULING DATA
BUDGETING DATA
SCHEDULING DATA
BUDGETING DATA
WORD PROCESSING PROGRAM
206
207
SCHEDULING DATA
BUDGETING DATA
EXPLANATORY DATA

Figure 2

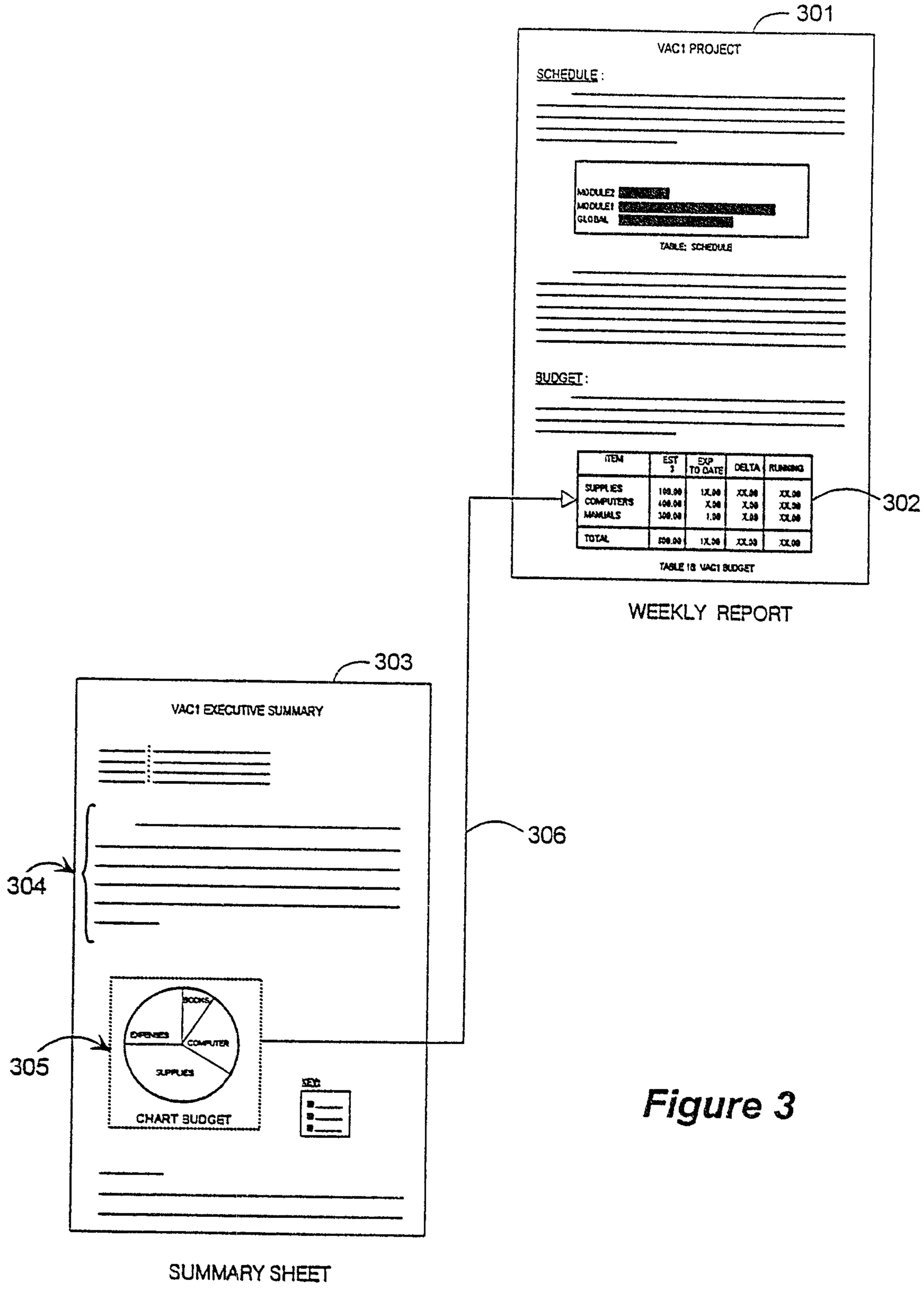

301
VAC1 PROJECT
SCHEDULE :
MODULE2
MODULE1
GLOBAL
TABLE: SCHEDULE
BUDGET :
ITEM
EST $
EXP TO DATE
DELTA
RUNNING
SUPPLIES
COMPUTERS
MANUALS
TOTAL
TABLE 18 VAC1 BUDGET
302
WEEKLY REPORT
303
VAC1 EXECUTIVE SUMMARY
304
305
306
BOOKS
EXPENSES
COMPUTER
SUPPLIES
CHART BUDGET
SUMMARY SHEET

Figure 3

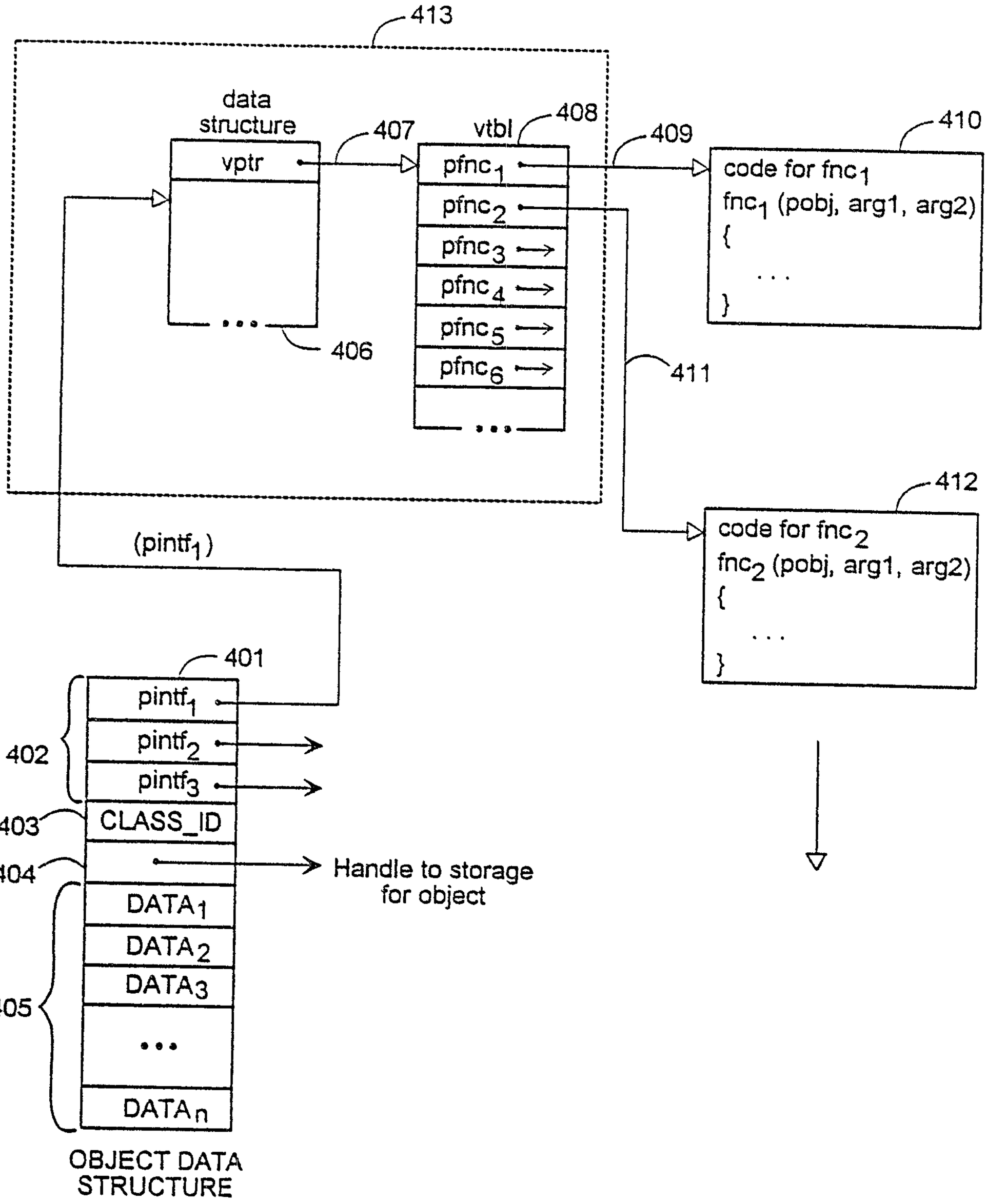

413
data structure
vptr
407
vtbl
408
409
410
pfnc1
pfnc2
pfnc3
pfnc4
pfnc5
pfnc6
406
code for fnc1
fnc1 (pobj, arg1, arg2)
{
...
}
411
412
code for fnc2
fnc2 (pobj, arg1, arg2)
{
...
}
(pintf1)
401
pintf1
pintf2
pintf3
402
403
CLASS_ID
404
Handle to storage for object
DATA1
DATA2
DATA3
405
DATAn
OBJECT DATA STRUCTURE

Figure 4

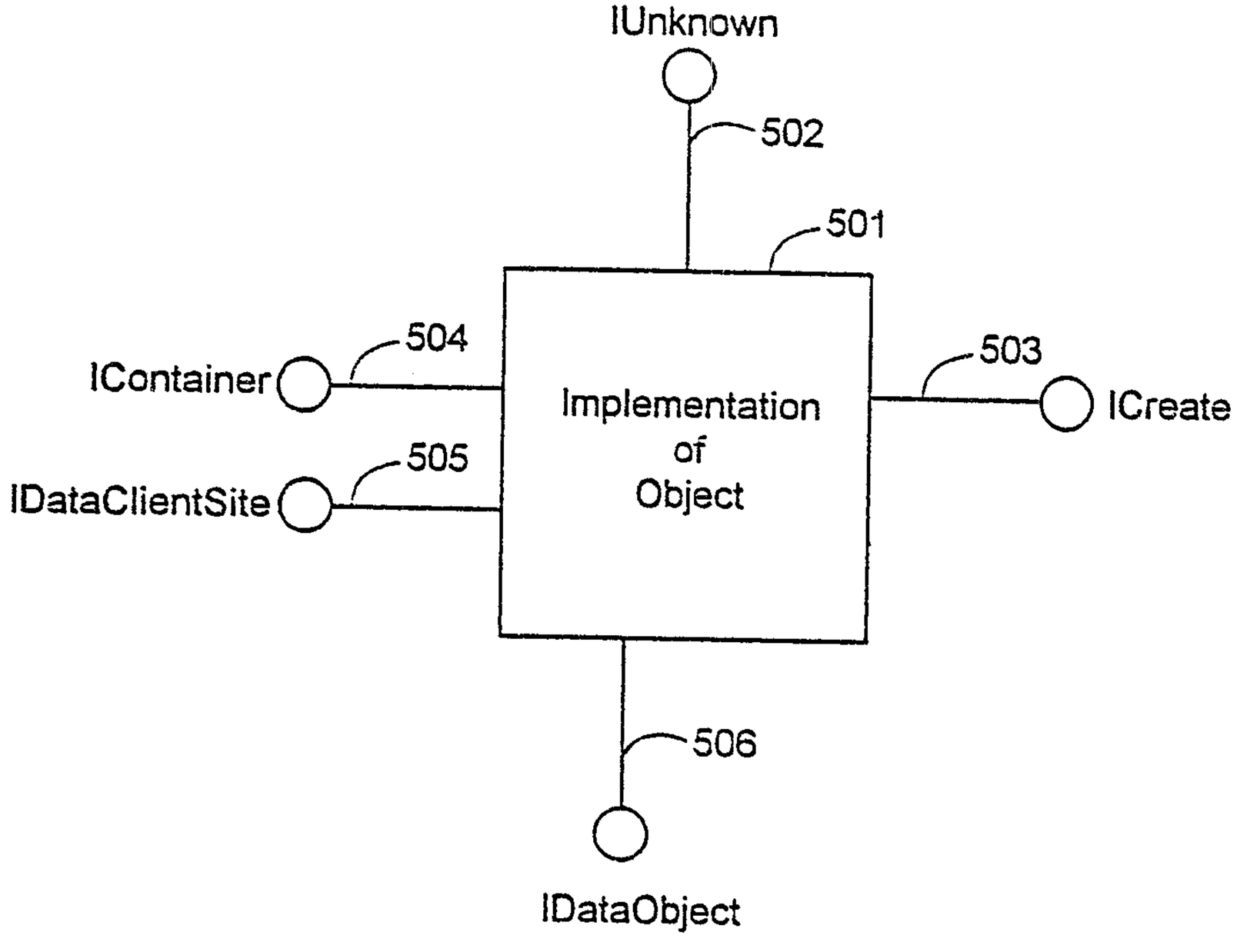

IUnknown
502
501
504
IContainer
503
ICreate
Implementation
of
Object
505
IDataClientSite
506
IDataObject

Figure 5

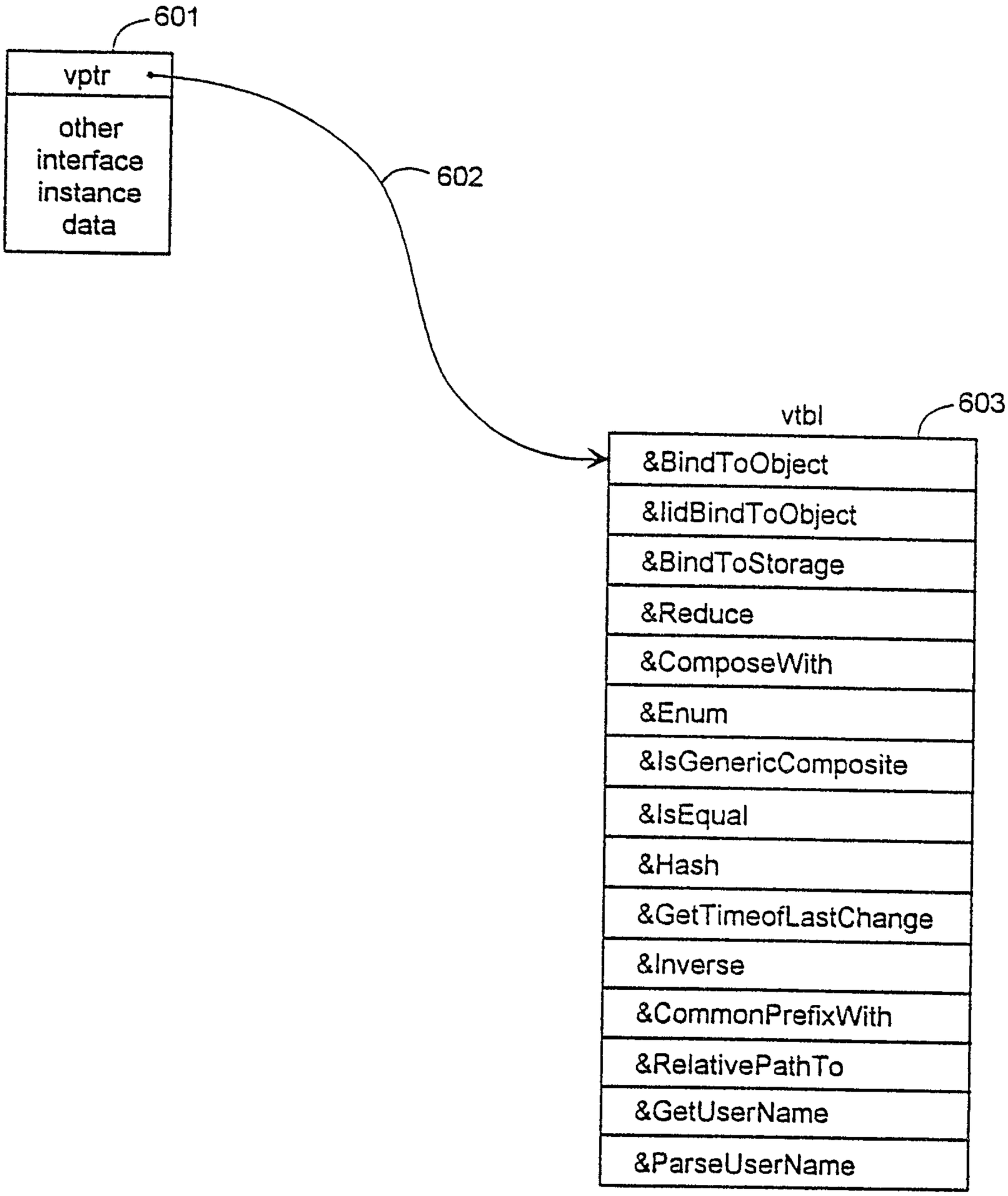

601
vptr
other
interface
instance
data
602
vtbl
603
&BindToObject
&IidBindToObject
&BindToStorage
&Reduce
&ComposeWith
&Enum
&IsGenericComposite
&IsEqual
&Hash
&GetTimeofLastChange
&Inverse
&CommonPrefixWith
&RelativePathTo
&GetUserName
&ParseUserName
IMONIKER ABSTRACT CLASS

Figure 6

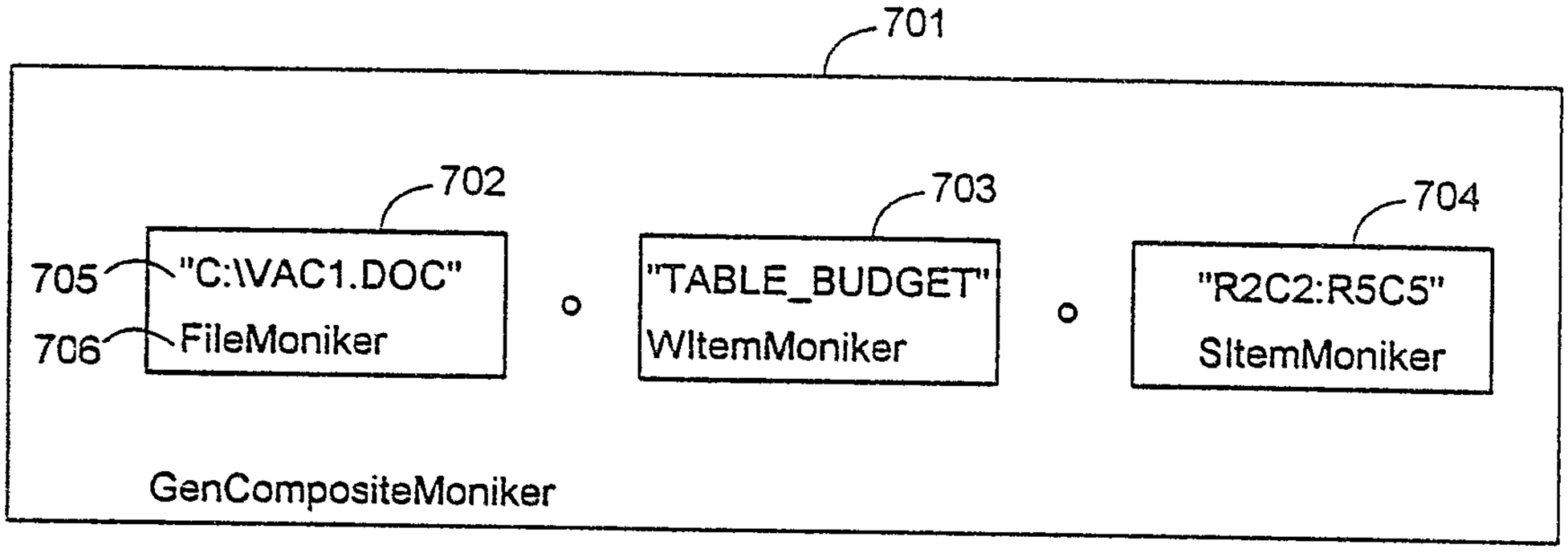

701
702
703
704
705
706
"C:\VAC1.DOC"
FileMoniker
"TABLE_BUDGET"
WItemMoniker
"R2C2:R5C5"
SItemMoniker
GenCompositeMoniker

Figure 7

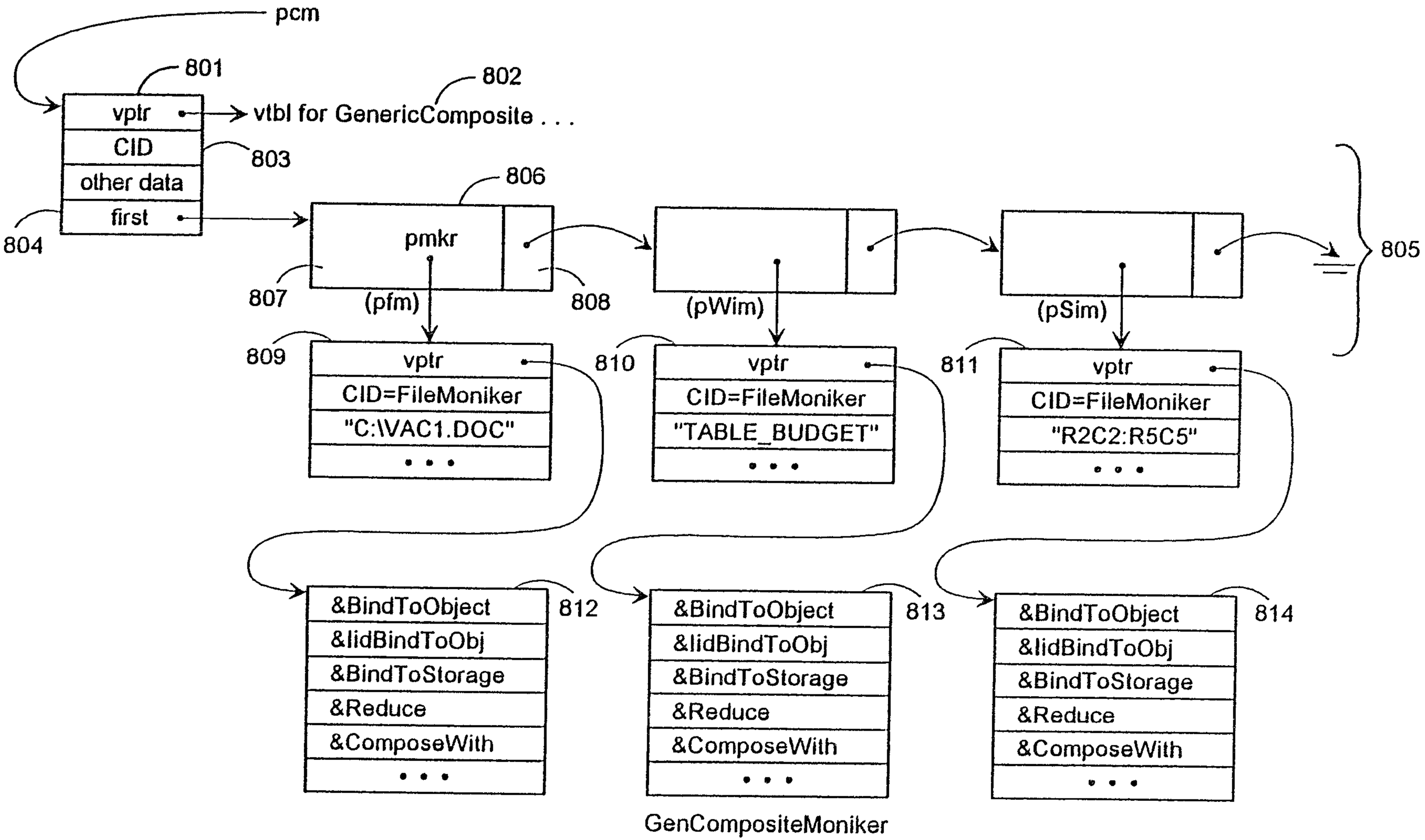


Figure 8

6682 9918 9318

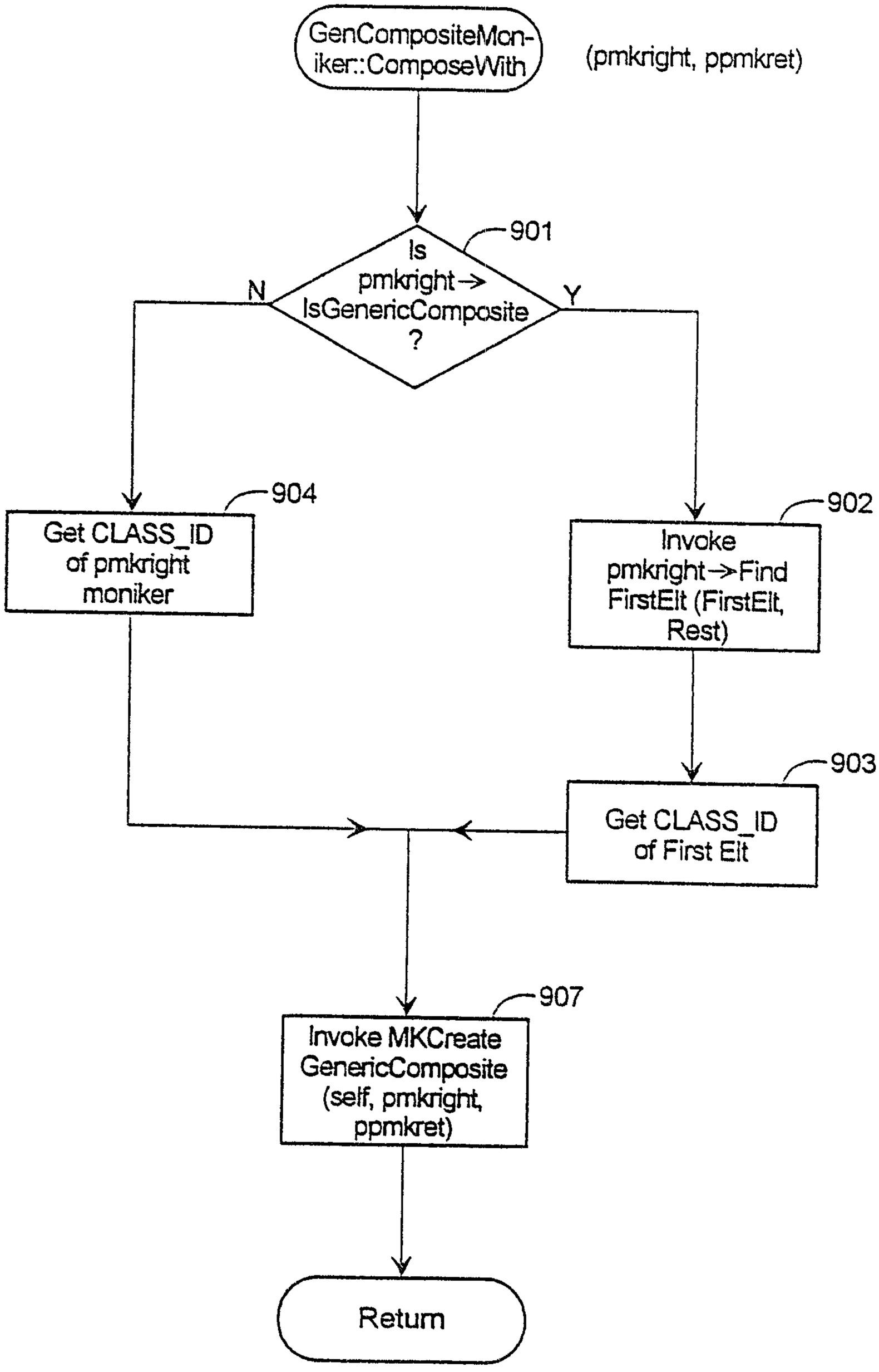

GenCompositeMon-iker::ComposeWith
(pmkright, ppmkret)
901
Is pmkright→ IsGenericComposite ?
N
Y
904
Get CLASS_ID of pmkright moniker
902
Invoke pmkright→Find FirstElt (FirstElt, Rest)
903
Get CLASS_ID of First Elt
907
Invoke MKCreate GenericComposite (self, pmkright, ppmkret)
Return

Figure 9

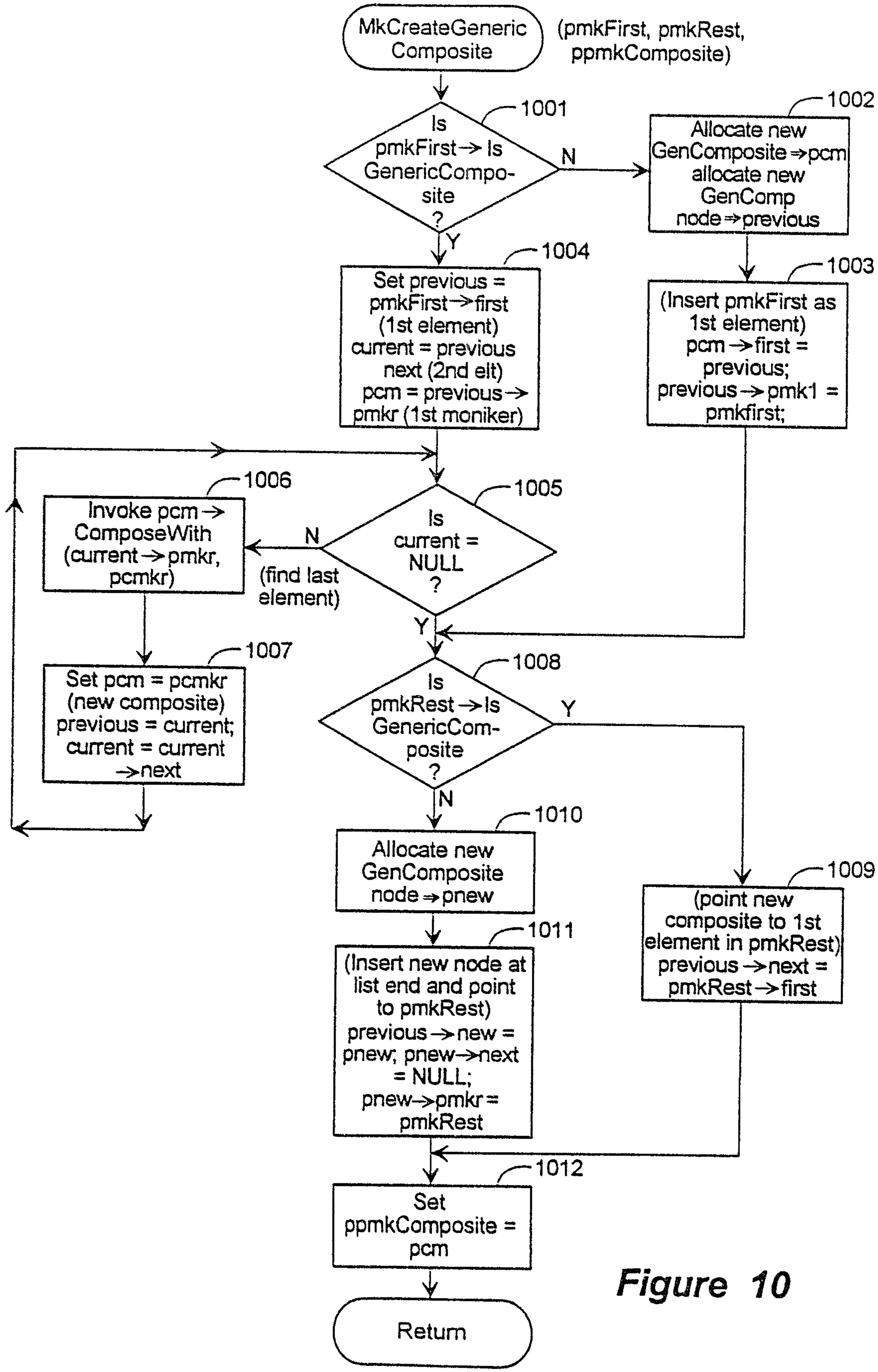

MkCreateGeneric Composite
(pmkFirst, pmkRest, ppmkComposite)
1001
Is pmkFirst→Is GenericCompo-site ?
N
Y
1002
Allocate new GenComposite⇒pcm allocate new GenComp node⇒previous
1003
(Insert pmkFirst as 1st element) pcm→first = previous; previous→pmk1 = pmkfirst;
1004
Set previous = pmkFirst→first (1st element) current = previous next (2nd elt) pcm = previous→ pmkr (1st moniker)
1005
Is current = NULL ?
N
(find last element)
Y
1006
Invoke pcm→ ComposeWith (current→pmkr, pcmkr)
1007
Set pcm = pcmkr (new composite) previous = current; current = current →next
1008
Is pmkRest→Is GenericCom-posite ?
Y
N
1009
(point new composite to 1st element in pmkRest) previous→next = pmkRest→first
1010
Allocate new GenComposite node⇒pnew
1011
(Insert new node at list end and point to pmkRest) previous→new = pnew; pnew→next = NULL; pnew→pmkr = pmkRest
1012
Set ppmkComposite = pcm
Return

Figure 10

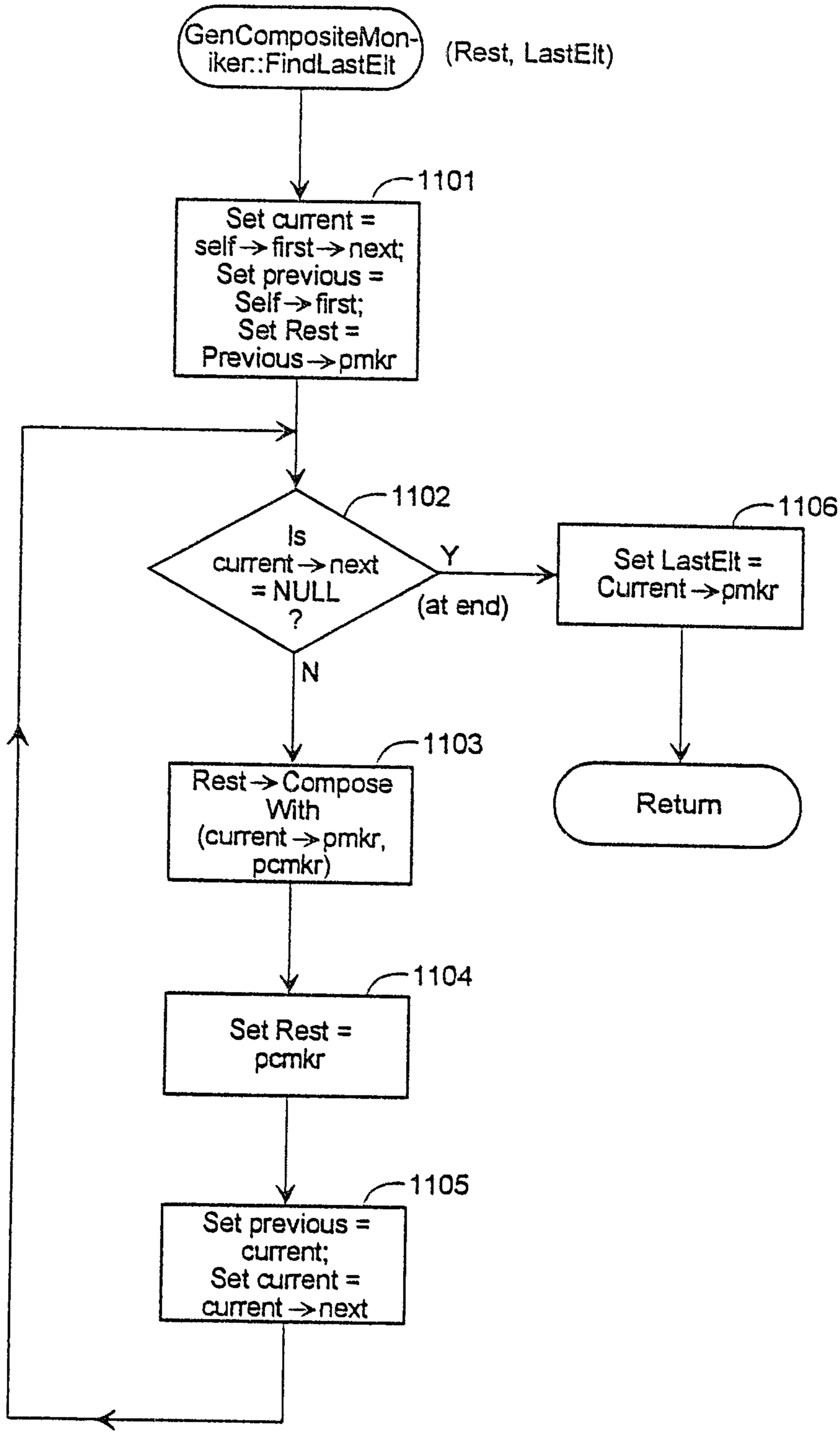

GenCompositeMon-iker::FindLastElt
(Rest, LastElt)
1101
Set current = self→first→next;
Set previous = Self→first;
Set Rest = Previous→pmkr
1102
Is current→next = NULL ?
Y
(at end)
1106
Set LastElt = Current→pmkr
Return
N
1103
Rest→Compose With (current→pmkr, pcmkr)
1104
Set Rest = pcmkr
1105
Set previous = current;
Set current = current→next

Figure 11

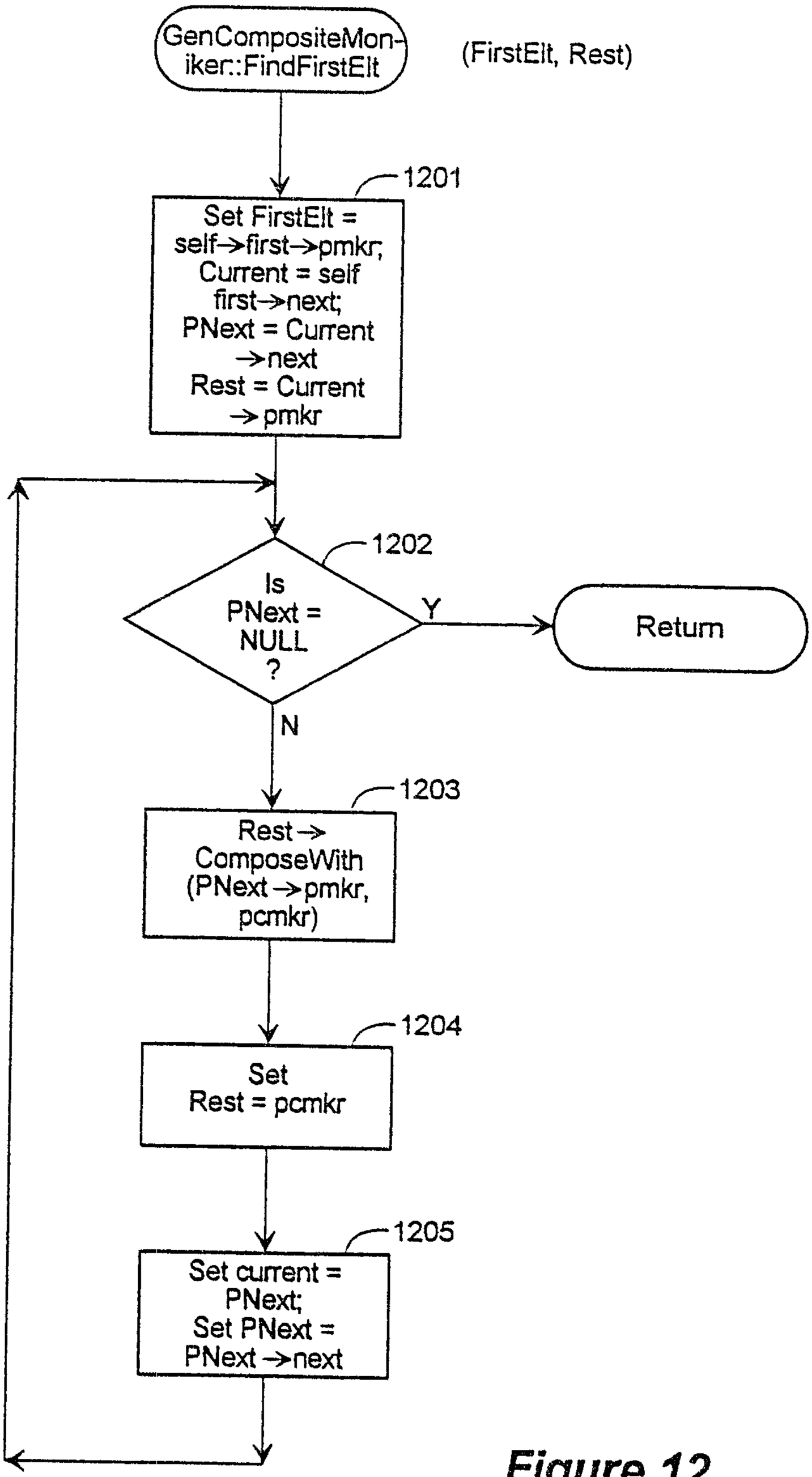

GenCompositeMoniker::FindFirstElt
(FirstElt, Rest)
1201
Set FirstElt = self→first→pmkr;
Current = self first→next;
PNext = Current →next
Rest = Current →pmkr
1202
Is PNext = NULL ?
Y
Return
N
1203
Rest→ ComposeWith (PNext→pmkr, pcmkr)
1204
Set Rest = pcmkr
1205
Set current = PNext;
Set PNext = PNext→next

Figure 12

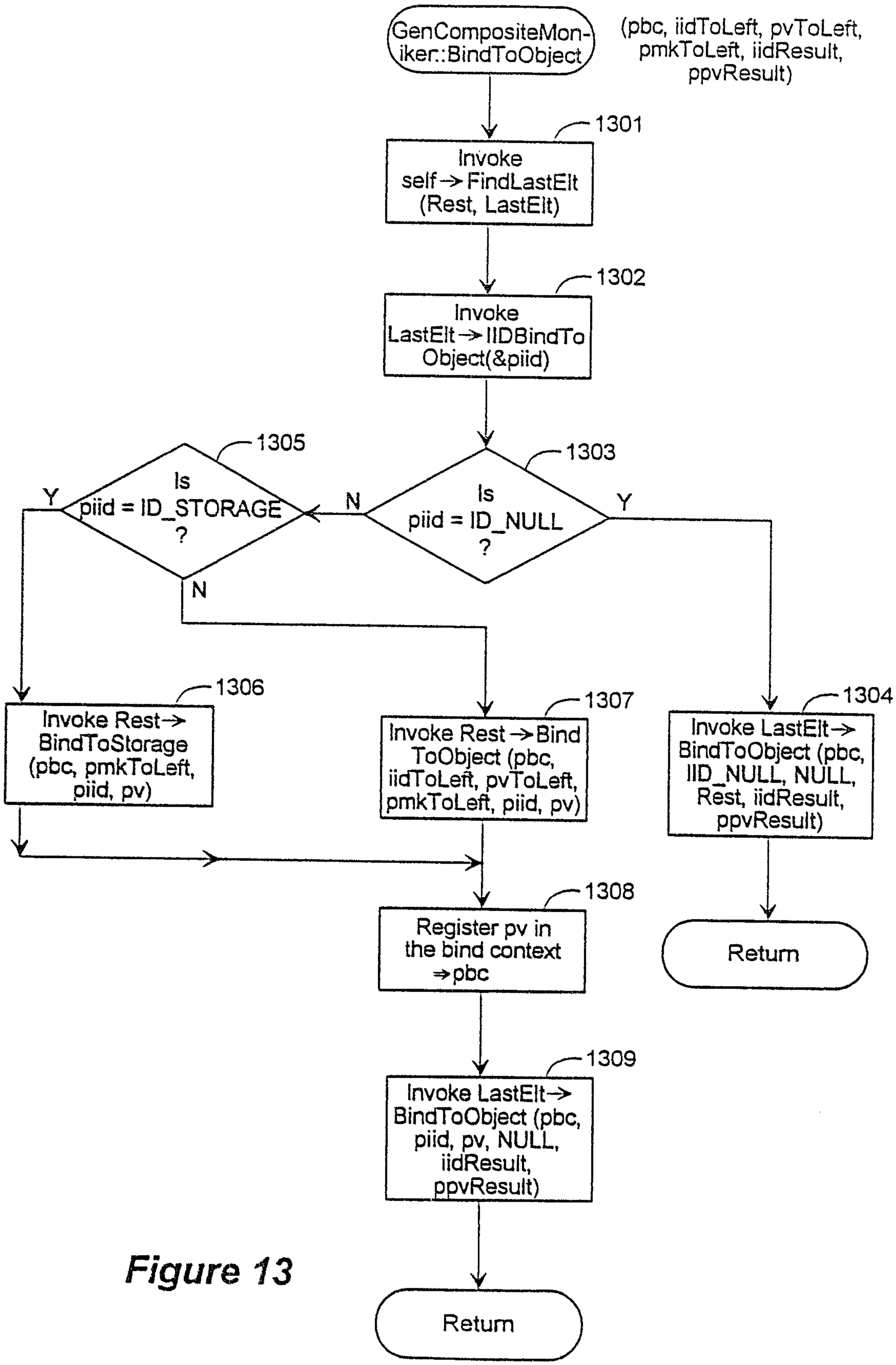

GenCompositeMoniker::BindToObject
(pbc, iidToLeft, pvToLeft, pmkToLeft, iidResult, ppvResult)
1301
Invoke self→FindLastElt (Rest, LastElt)
1302
Invoke LastElt→IIDBindTo Object(&piid)
1303
Is piid = ID_NULL ?
Y
N
1305
Is piid = ID_STORAGE ?
Y
N
1304
Invoke LastElt→ BindToObject (pbc, IID_NULL, NULL, Rest, iidResult, ppvResult)
Return
1306
Invoke Rest→ BindToStorage (pbc, pmkToLeft, piid, pv)
1307
Invoke Rest→Bind ToObject (pbc, iidToLeft, pvToLeft, pmkToLeft, piid, pv)
1308
Register pv in the bind context →pbc
1309
Invoke LastElt→ BindToObject (pbc, piid, pv, NULL, iidResult, ppvResult)
Return

Figure 13

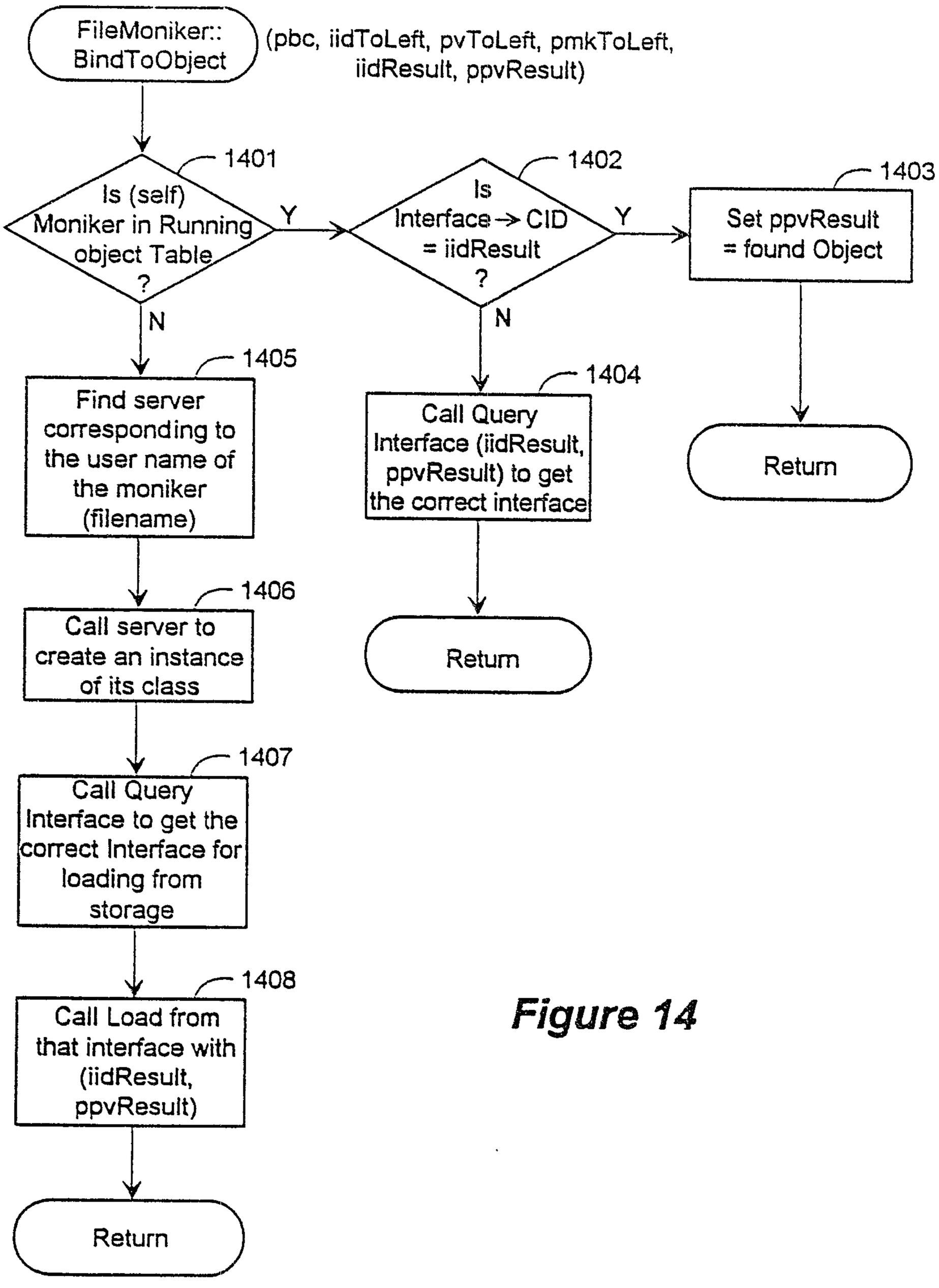

FileMoniker::
BindToObject
(pbc, iidToLeft, pvToLeft, pmkToLeft,
iidResult, ppvResult)
1401
Is (self)
Moniker in Running
object Table
?
Y
N
1402
Is
Interface → CID
= iidResult
?
Y
N
1403
Set ppvResult
= found Object
Return
1404
Call Query
Interface (iidResult,
ppvResult) to get
the correct interface
Return
1405
Find server
corresponding to
the user name of
the moniker
(filename)
1406
Call server to
create an instance
of its class
1407
Call Query
Interface to get the
correct Interface for
loading from
storage
1408
Call Load from
that interface with
(iidResult,
ppvResult)
Return

Figure 14

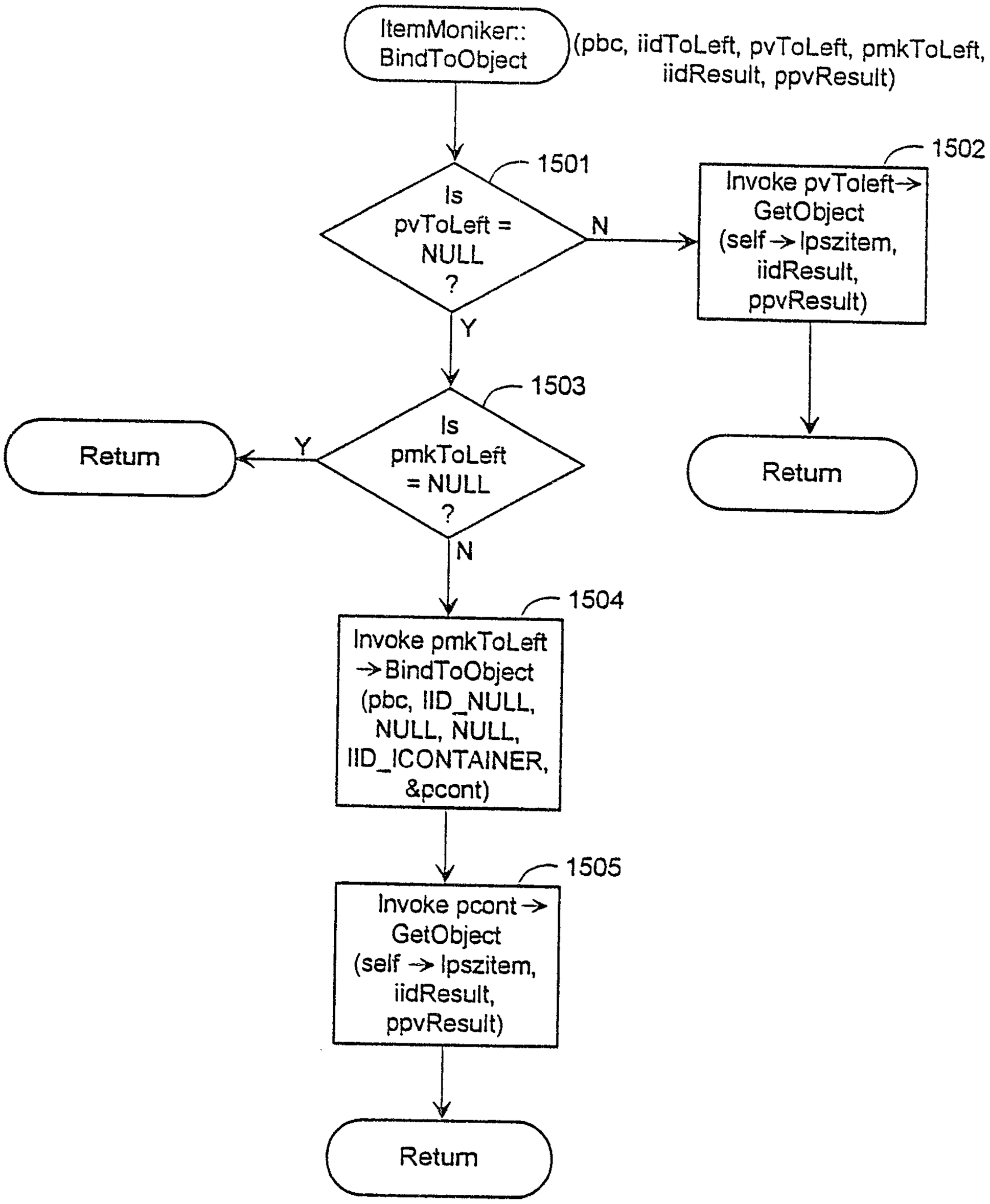

ItemMoniker::
BindToObject
(pbc, iidToLeft, pvToLeft, pmkToLeft,
iidResult, ppvResult)
1501
Is
pvToLeft =
NULL
?
N
Y
1502
Invoke pvToleft→
GetObject
(self→lpszitem,
iidResult,
ppvResult)
Return
1503
Is
pmkToLeft
= NULL
?
Y
N
Return
1504
Invoke pmkToLeft
→BindToObject
(pbc, IID_NULL,
NULL, NULL,
IID_ICONTAINER,
&pcont)
1505
Invoke pcont→
GetObject
(self→lpszitem,
iidResult,
ppvResult)
Return

Figure 15

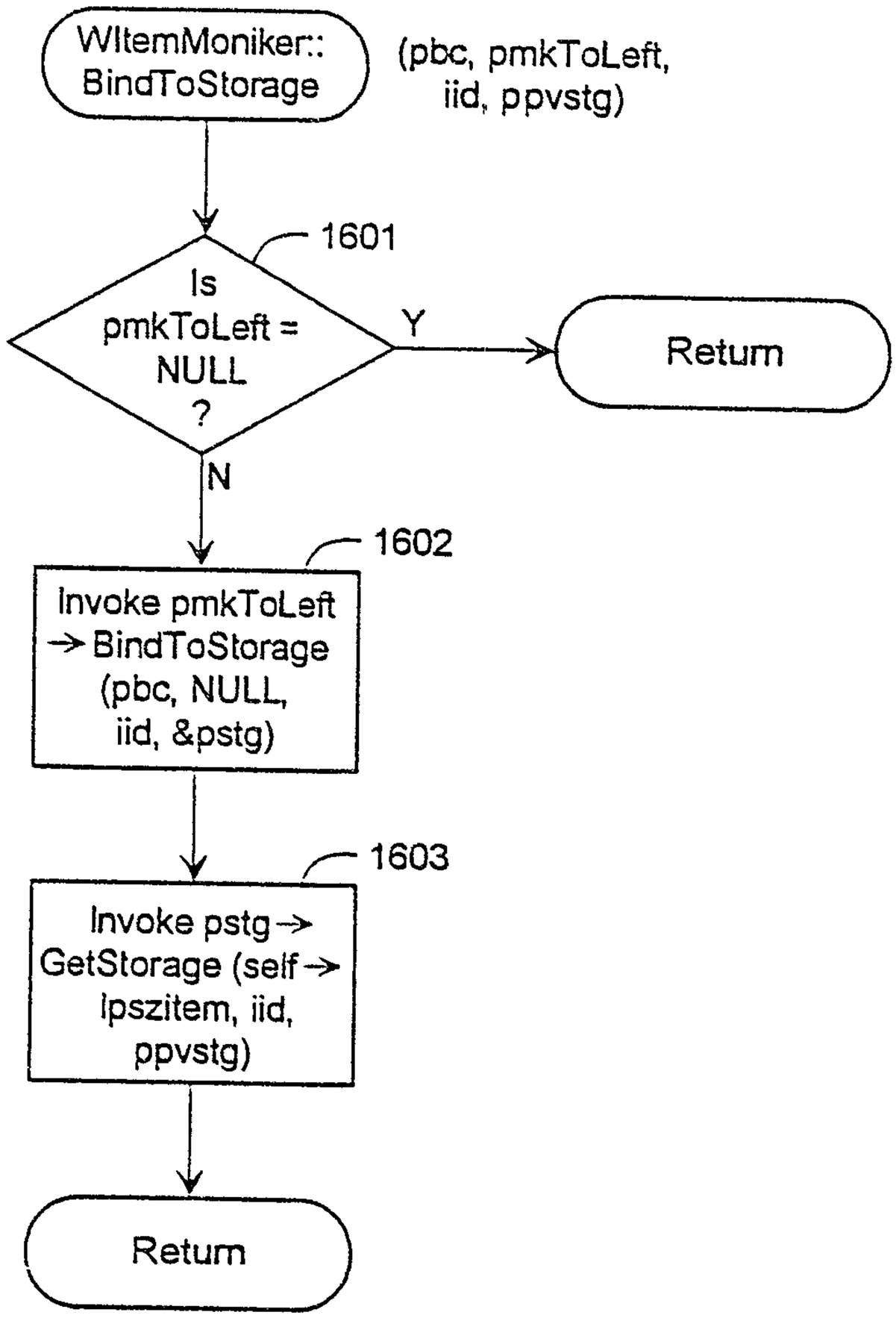

WItemMoniker::
BindToStorage
(pbc, pmkToLeft,
iid, ppvstg)
1601
Is
pmkToLeft =
NULL
?
Y
Return
N
1602
Invoke pmkToLeft
→ BindToStorage
(pbc, NULL,
iid, &pstg)
1603
Invoke pstg→
GetStorage (self→
lpszitem, iid,
ppvstg)
Return

Figure 16

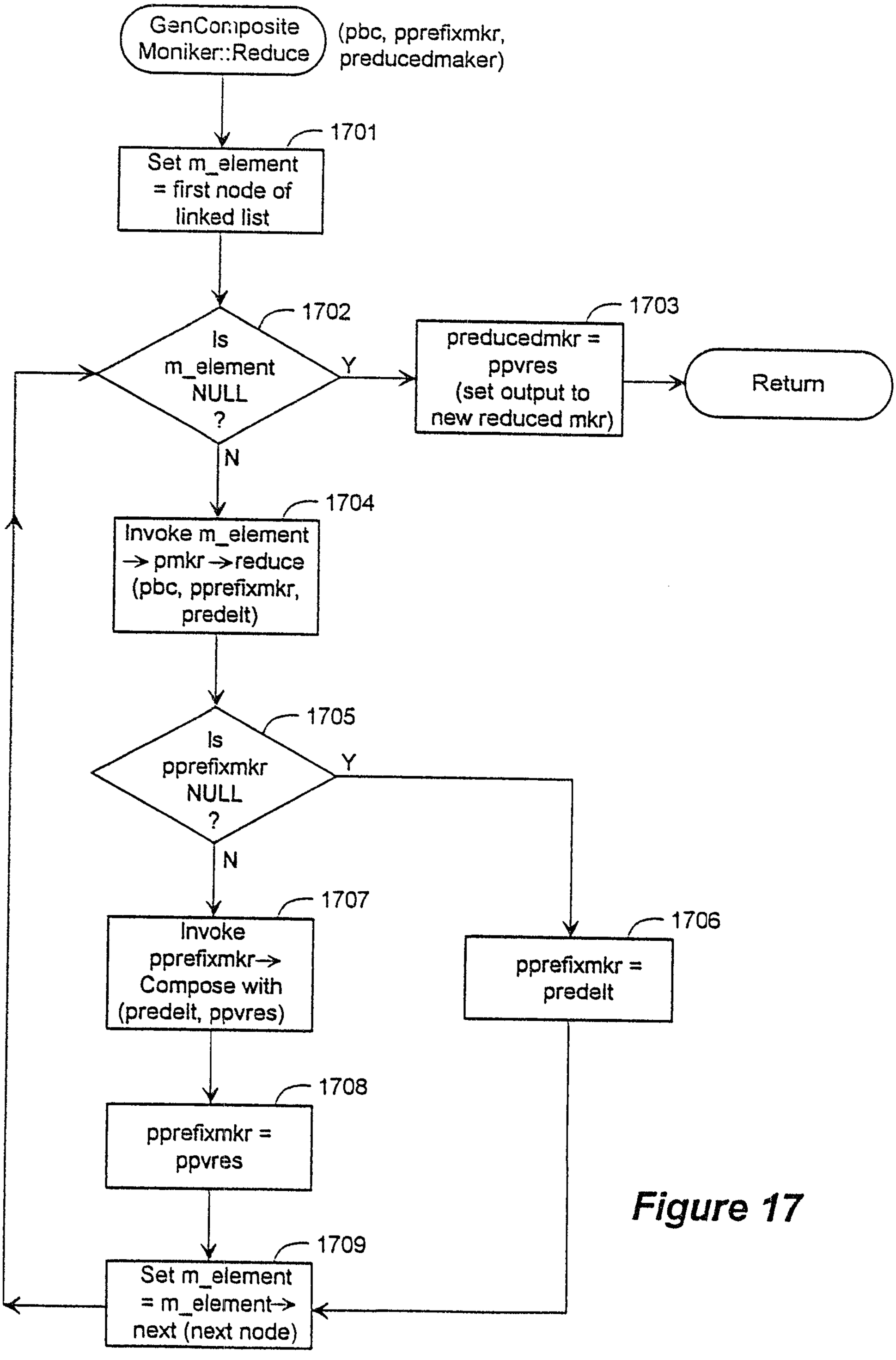

GenComposite Moniker::Reduce
(pbc, pprefixmkr, preducedmaker)
1701
Set m_element = first node of linked list
1702
Is m_element NULL ?
Y
1703
preducedmkr = ppvres (set output to new reduced mkr)
Return
N
1704
Invoke m_element → pmkr → reduce (pbc, pprefixmkr, predelt)
1705
Is pprefixmkr NULL ?
Y
N
1706
pprefixmkr = predelt
1707
Invoke pprefixmkr→ Compose with (predelt, ppvres)
1708
pprefixmkr = ppvres
1709
Set m_element = m_element→ next (next node)

Figure 17

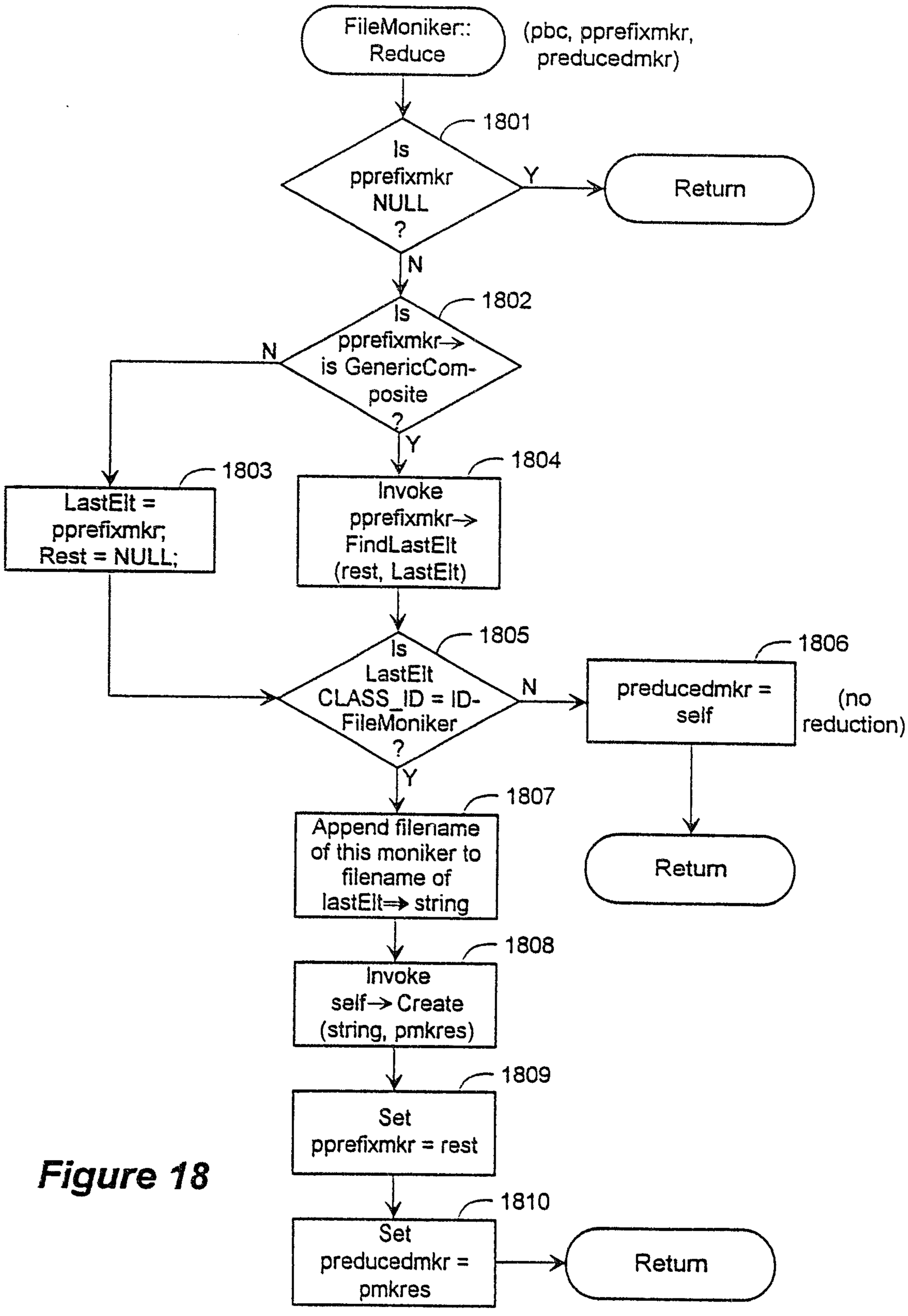

FileMoniker::
Reduce
(pbc, pprefixmkr,
preducedmkr)
1801
Is
pprefixmkr
NULL
?
Y
Return
N
1802
Is
pprefixmkr→
is GenericCom-
posite
?
N
Y
1803
LastElt =
pprefixmkr;
Rest = NULL;
1804
Invoke
pprefixmkr→
FindLastElt
(rest, LastElt)
1805
Is
LastElt
CLASS_ID = ID-
FileMoniker
?
N
Y
1806
preducedmkr =
self
(no
reduction)
Return
1807
Append filename
of this moniker to
filename of
lastElt→ string
1808
Invoke
self→ Create
(string, pmkres)
1809
Set
pprefixmkr = rest
1810
Set
preducedmkr =
pmkres
Return

Figure 18

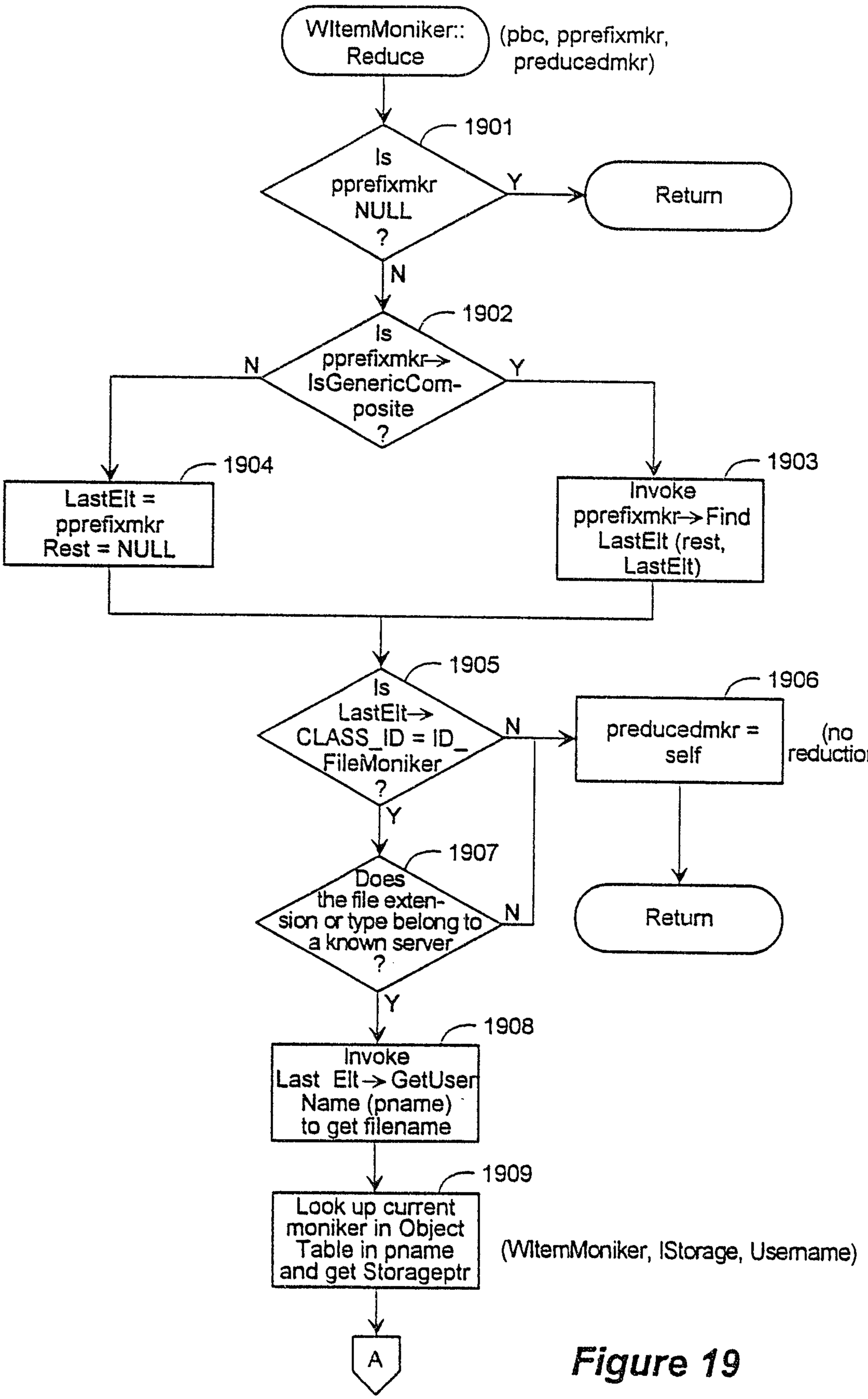

WItemMoniker::
Reduce
(pbc, pprefixmkr,
preducedmkr)
1901
Is
pprefixmkr
NULL
?
Y
Return
N
1902
Is
pprefixmkr→
IsGenericCom-
posite
?
N
Y
1904
LastElt =
pprefixmkr
Rest = NULL
1903
Invoke
pprefixmkr→Find
LastElt (rest,
LastElt)
1905
Is
LastElt→
CLASS_ID = ID_
FileMoniker
?
N
1906
preducedmkr =
self
(no
reduction)
Y
1907
Does
the file exten-
sion or type belong to
a known server
?
N
Return
Y
1908
Invoke
Last Elt→GetUser
Name (pname)
to get filename
1909
Look up current
moniker in Object
Table in pname
and get Storageptr
(WItemMoniker, IStorage, Username)
A

Figure 19

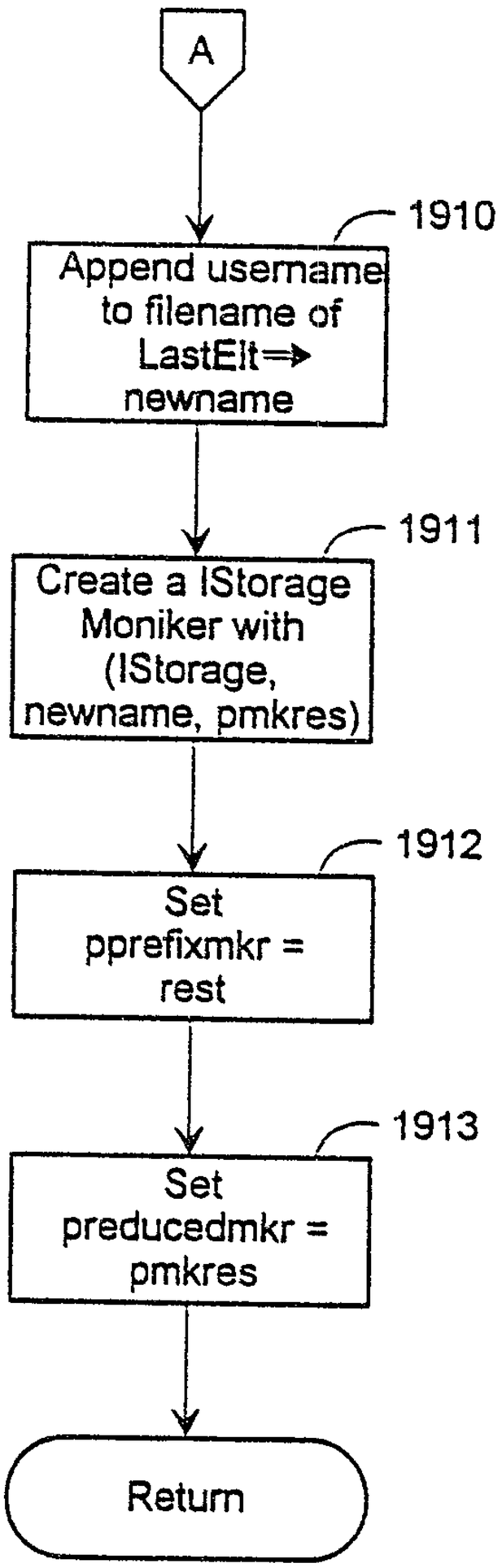


Figure 19 (Cont'd)

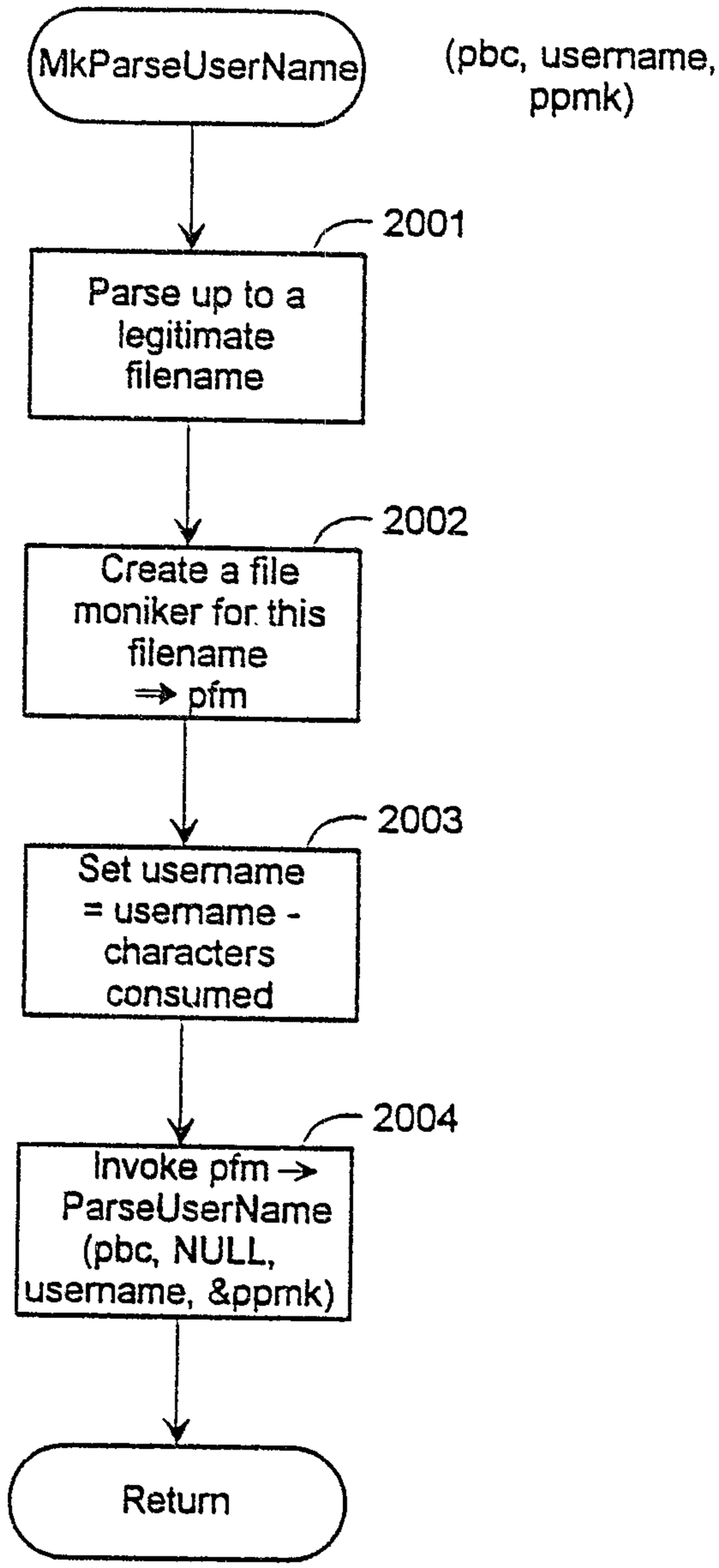

MkParseUserName
(pbc, username, ppmk)
2001
Parse up to a legitimate filename
2002
Create a file moniker for this filename ⇒ pfm
2003
Set username = username - characters consumed
2004
Invoke pfm → ParseUserName (pbc, NULL, username, &ppmk)
Return

Figure 20

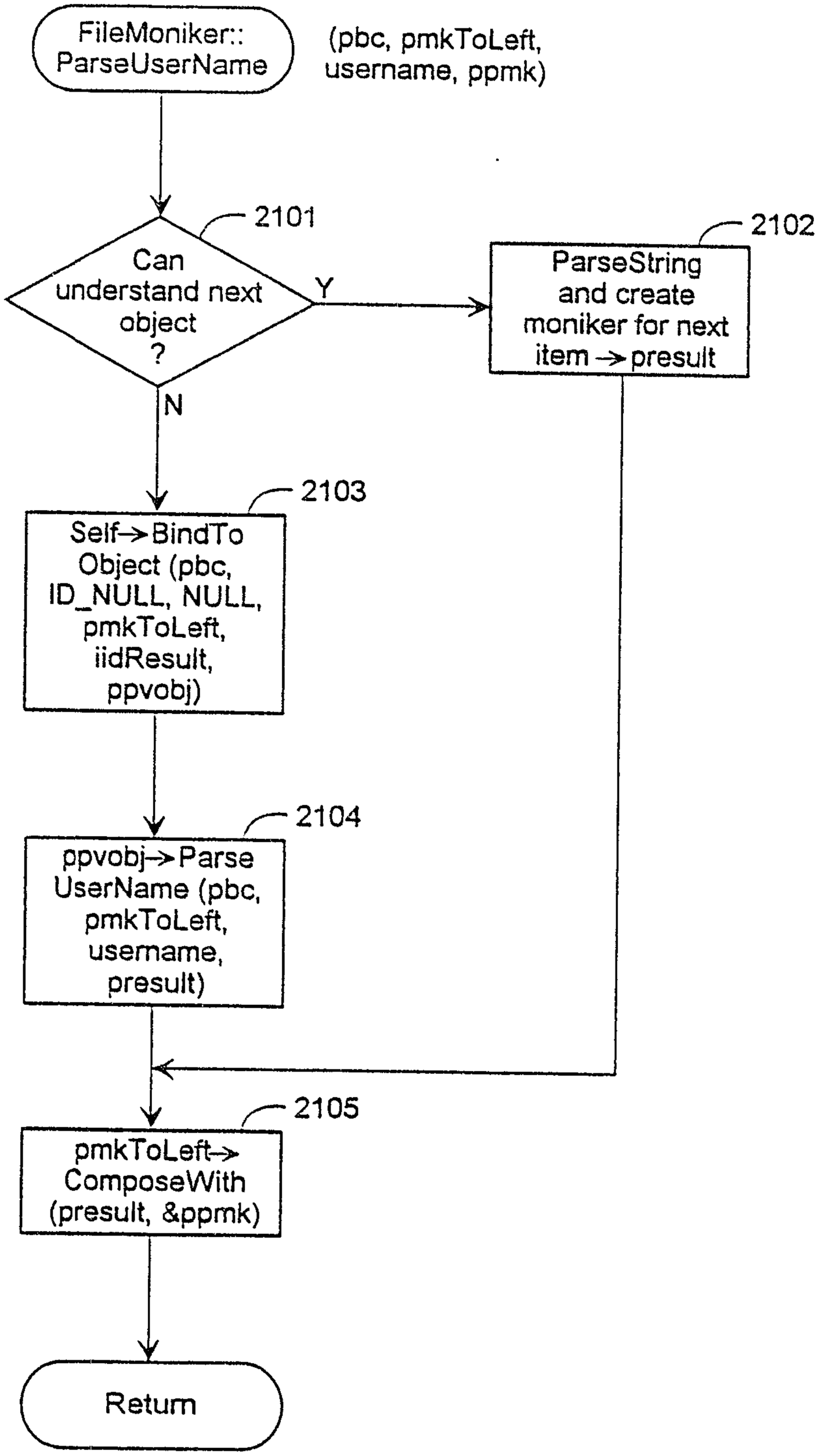

FileMoniker::
ParseUserName
(pbc, pmkToLeft,
username, ppmk)
2101
Can
understand next
object
?
Y
N
2102
ParseString
and create
moniker for next
item → presult
2103
Self→BindTo
Object (pbc,
ID_NULL, NULL,
pmkToLeft,
iidResult,
ppvobj)
2104
ppvobj→Parse
UserName (pbc,
pmkToLeft,
username,
presult)
2105
pmkToLeft→
ComposeWith
(presult, &ppmk)
Return

Figure 21

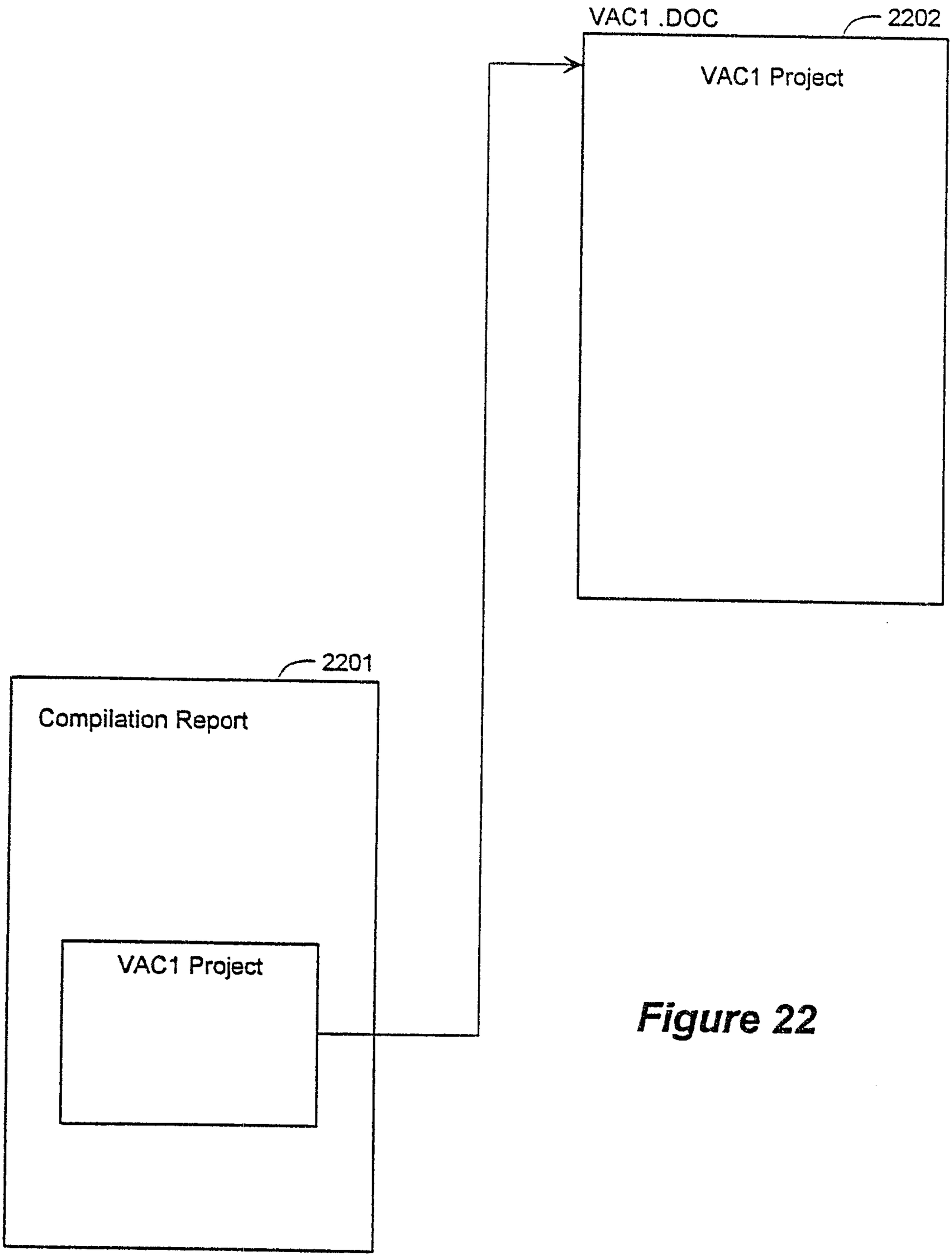
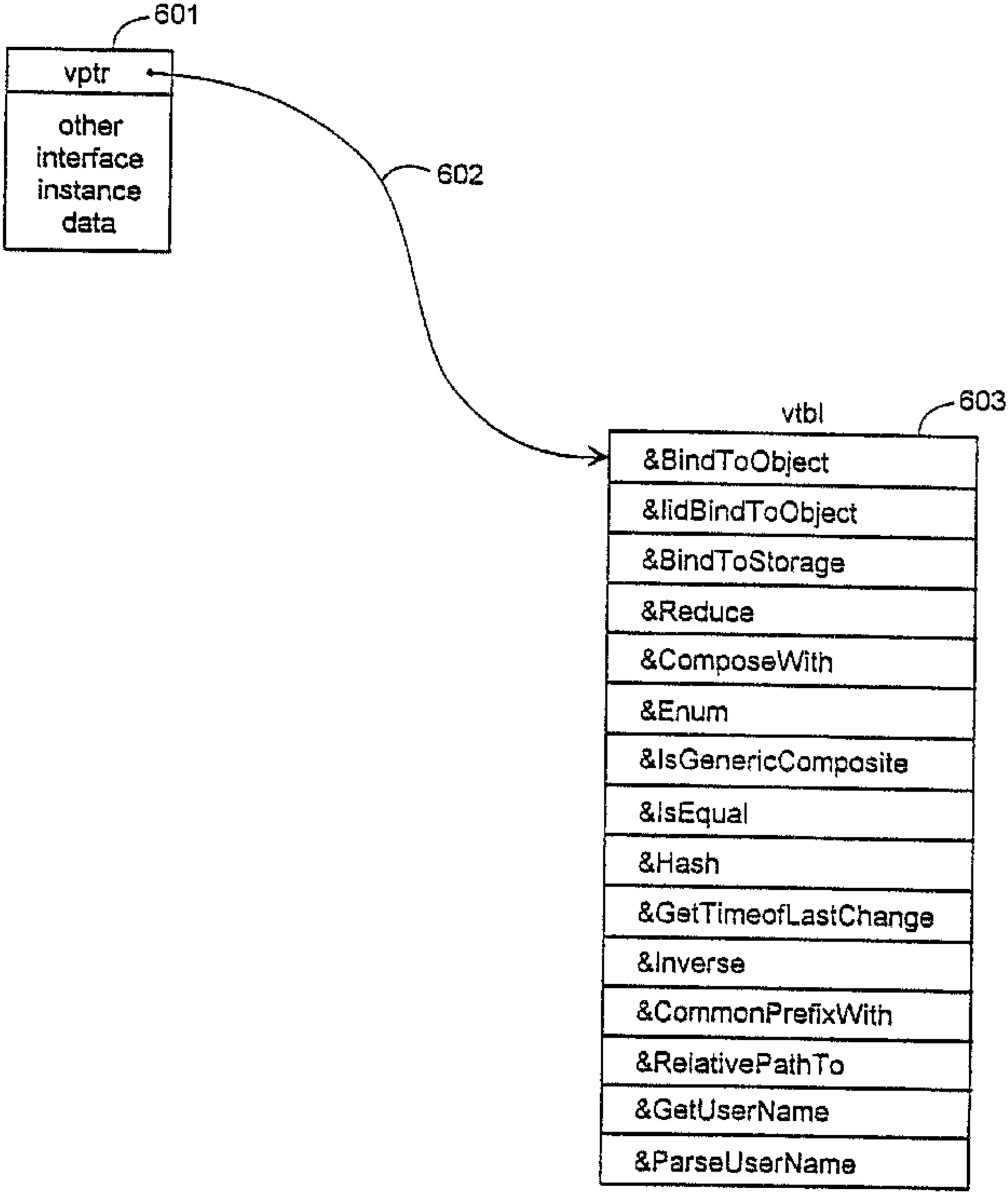

VAC1 .DOC
2202
VAC1 Project
2201
Compilation Report
VAC1 Project

Figure 22



IMONIKER ABSTRACT CLASS