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

Canadian Intellectual Property Office
An agency of Industry Canada

CA 2388101 A1 2003/08/01

(21) 2 388 101

(12) DEMANDE DE BREVET CANADIEN
CANADIAN PATENT APPLICATION

(13) A1

(22) Date de dépôt/Filing Date: 2002/05/30

(41) Mise à la disp. pub./Open to Public Insp.: 2003/08/01

(30) Priorité/Priority: 2002/02/01 (2,369,854) CA

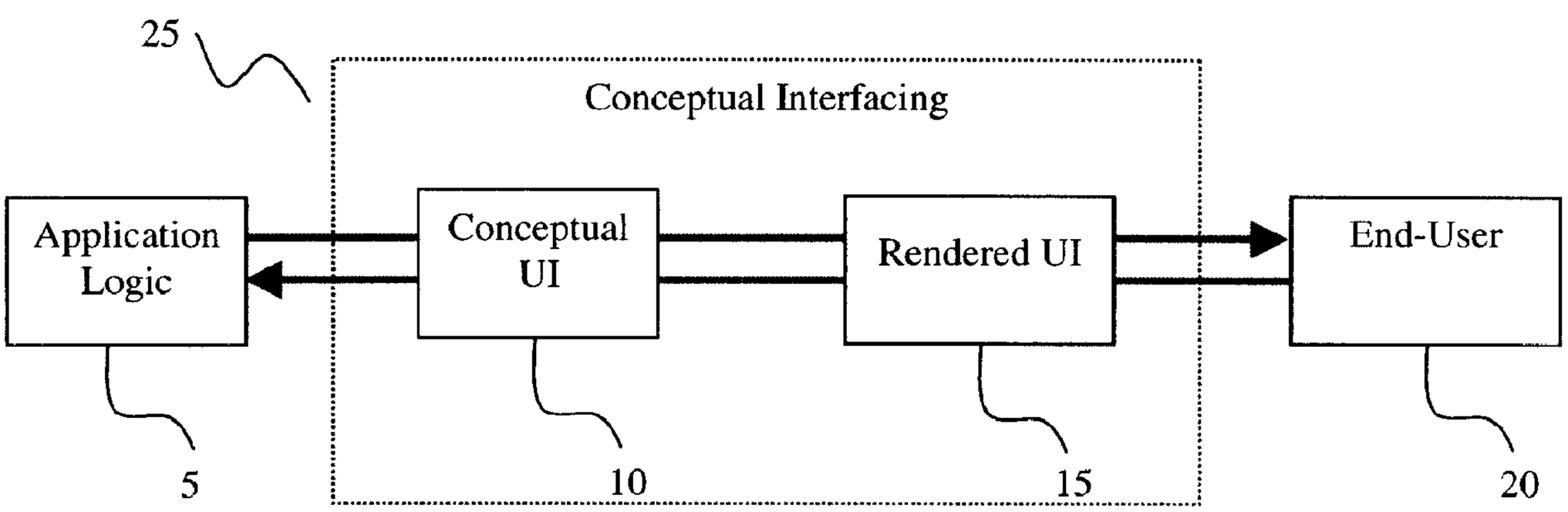
(51) Cl.Int.^{7}/Int.Cl.^{7} G06F 3/14

(71) Demandeur/Applicant:
CONCEPTS EGERIA INC., CA

(72) Inventeur/Inventor:
SAVAGE, MARTIN, CA

(74) Agent: TORYS LLP

(54) Titre : INTERFACE UTILISATEUR CONCEPTUELLE
(54) Title: CONCEPTUAL USER INTERFACE



1. Background of the invention

1.1 Field of the invention

The present invention describes a new methodology of designing and programming user interfaces. According to this new methodology, any user interface can be designed by using only three conceptual component types.

1.2 Background art

The traditional way of designing user interfaces is by using specific components related to the development platform. The development platform can be a specific Operating System platform (e.g. Windows, Linux, Mac...) or a specific device (PDA, cell phone). For each of these platforms, a hierarchy of proprietary components is used, creating an incompatibility between each other. These components are usually related to visual capacities and specific capabilities of the target platform (components such as windows and buttons are common on GUIs whereas small menus are used on wireless devices), extending the incompatibility between GUIs and wireless interfaces. Moreover, they have inherent limitations: remote access is not available without external software and there is no multi-user collaboration functionality implemented.

Java was the best attempt to overcome the GUI incompatibility by offering its adaptable set of proprietary components. However, Java still offers at least three sets of UI components: AWT / SWING for GUIs, J2ME for wireless devices and JTAPI for vocal access. It also doesn't really run remotely since applets must be downloaded first before running locally on the client.

So up to now, none of the existing user interface components is really portable on different platforms creating a necessity of redesigning and recoding to support other existing platforms. All these incompatibilities between different user interfaces are based on the following statement from the J2ME white paper:

"Consumer devices have substantial differences in memory size, networking, and user interface capabilities, making it very difficult to support all devices with just one solution."

1.3 Summary of the invention

The objective of the present invention is to provide one solution that can support all devices whatever their differences are in memory size, networking and/or user interface capabilities. According to this method, users interfaces are no longer designed using specific components for specific target platforms; they are designed using three generic conceptual user interface components.

Using the generic conceptual components (subject of this invention), user interfaces are designed according to the needs of the user not to the capabilities of a specific platform. The look and feel is rendered at run-time and depends on the targeted platform or device.

This methodology doesn't only solve the problem of incompatibilities between the user interfaces of different platforms but also provides a new way of designing a user interface without having to think of where it will be deployed. The generic aspect and natural portability of the conceptual user interface generates a substantial relief for the software developers who will not be anymore obliged to understand the capabilities of the different platforms or devices. Instead, they will focus more on designing and defining the functionalities of the application than on accommodating it to run on specific platforms.

According to its conceptual nature, this invention does not only solve all the user interface incompatibilities for the existing platforms and devices, it will accommodate any substantial future platform or device. To finish, this invention does definitely create a new way of designing or even more defining user interfaces, a way that will make it possible for all devices to interact between each other.

2. Brief description of drawings

FIG. 1 shows the conceptual interfacing process that occurs between the application logic and the end-user;

FIG. 2 presents the conceptual user interface stack;

FIG. 3 presents the inheritance diagram of the UI conceptual components;

FIG. 4 shows the general data structure and the modification methods of the UI conceptual component;

FIG. 5A, FIG. 5B and FIG. 5C respectively describe the data structure and modification methods of the container, control and text conceptual components;

FIG. 6 illustrates the meaning of the term Values Matrix for each Conceptual Component;

FIG. 7A shows a example of a tree of Conceptual Components (CCs); FIG. 7B shows the Rendered UI of a Container Conceptual Component with two child Conceptual Components;

FIG. 8 shows the different d-sizes (Base, Labels Matrix, States Matrix, Container, Values Matrix and Read-Only Flags Matrix);

FIG. 9 shows the effect of Constant Flags on the insert/remove methods;

FIG. 10 shows in a real example the interaction between the Selections Matrix and the States Matrix;

FIG. 11 shows, for a Control conceptual component, some possible renderings calculated from different combinations of States Matrix d-sizes, Bitmap Size and String Max Length values;

FIG. 12A and FIG. 12B illustrate in a real example of the Children Direction influence on the rendering of a Container CC;

FIG. 13 illustrates MDI windows that occur when End-User Resize flag of a 1-dimensioned CC is "true".

FIG. 14A and FIG. 14B show the conversion of an n-dimensional coordinate to a 3-dimensional (Cartesian) system; FIG. 14C shows the reverse conversion from 3-dimensional to n-dimensional system.

FIG. 15A shows the difference between the Matrix view and the element view; FIG. 15B shows the calculation rules of the element view for all CC types; FIG. 15C shows the User View for the Blackberry.

FIG. 16A shows the two steps for rendering a Conceptual Component into Native Components; FIG. 16B shows that different device components (DCs) can be rendered using common native components.

FIG. 17 shows the state machine composed of Windows device components as states, and their possible transitions when creating a device component tree.

FIG. 18A to FIG. 18H show, using flowcharts, the DC tree construction algorithm for the Windows GUI.

FIG. 19A to FIG. 19E show, using flowcharts, the NC construction algorithms for the Windows GUI.

FIG. 20 shows an example of the rendering of a Calculator Conceptual UI using Windows GUI.

FIG. 21 shows the DC tree navigation principle applicable to platforms with limited user interface capabilities.

3. Description of the Invention

3.1 The Conceptual Interfacing Process

With reference initially to FIG. 1, schematically shown therein is the flow of information between the Application Logic **5** and an End-User **20**. Within the Application Logic **5**, the Conceptual UI **10** is created by using Conceptual Components (explained later). The Conceptual UI **10** is then converted into a Rendered UI **15**, using algorithms proprietary to the targeted client platform/device, to be finally interpreted by the End-User **20**. Conversely, the End-User **20** may modify the Application Logic **5** by applying actions on it. The Conceptual Interfacing **25** methodology, subject of this invention encloses the Conceptual UI **10** and the Rendered UI **15**.

To understand this Conceptual Interfacing **25** process, three different parts are to be discussed in details:

1. The Conceptual UI **10** that is a stack of references to UI Conceptual Components.
2. The Conceptual Components needed to build a complete Conceptual UI according to this invention.

3. The Rendered UI **15** is a presentable version of the Conceptual UI **10**, by generating in real-time a visible and interactive user-interface according to the specifications and capabilities of the End-User **20** target platform/device.

3.2 Conceptual UI

A Conceptual UI is a user-interface model composed of abstract components such as containers, controls and texts instead of device-specific components such as windows, forms, buttons, menus, check boxes and edit boxes. To be able to conceptualize any user interface into a few abstract components (three according to this invention), a mathematical model had to be used. This mathematical model includes multi-dimensioned numbers, vectors, sets, matrices, trees and matrices of trees. Only mathematically proven concepts made possible the transition from a device-capabilities-based model (such as Win32, AWT, RIMOS) to a fully user-need-based model (Conceptual UI).

Referring to FIG. 2, the implementation of a Conceptual UI is a simple Stack **12** of references to conceptual components, detailed in the next section. References in the stack may be null as well (the reason is explained later). Both stack and conceptual components are entirely managed by the application logic. The application logic, created by the programmer, can create conceptual components by defining them and calling their constructors. He can then add the references of the newly created conceptual components to the Conceptual UI Stack **12** via the *Push(CC)* **65** method, and can later remove existing references from the Conceptual UI Stack **12** by calling the *Pop()* **70** method. The push and pop methods will be explained in details shortly.

Consider the Conceptual Component for now as the base unit defining user-interface information presented to the user. When a Conceptual Component is created in the application, it is not immediately rendered to the user. The rendering of this Conceptual Component to the user happens only when the programmer adds the conceptual component's reference to the Conceptual UI Stack **12**. Once a Conceptual Component is referenced in the Conceptual UI Stack **12**, it is considered "linked". A Conceptual Component not referenced in the Conceptual UI Stack **12** is considered "unlinked". Referring to FIG. 2, the Conceptual Component **50a** and the Conceptual Component **50b** are linked, whereas the Conceptual Component **50c** is unlinked.

At application initialization, the Conceptual UI Stack **12** is always empty. Like any stack, the Conceptual UI has two methods: *Push(CC)* **65** and *Pop()* **70**.

The *Push(CC)* **65** will add a reference, at the top of the stack, that points to either a valid Conceptual Component or null. When a new Conceptual Component is pushed in the Conceptual UI Stack **12**, it becomes linked and the previously pushed Conceptual Component (if any) becomes read-only. User interaction always takes place only on the Conceptual Component located at the top of the stack, even if all linked Conceptual Components are rendered. When the null value is pushed in the Conceptual UI Stack **12**, no user interaction is allowed for any linked Conceptual Components, until either *Pop()* **70** is called or a new Conceptual Component is pushed.

The *Pop()* **70** method removes the last pushed reference from the top of the stack. User interaction is redirected to the new top Conceptual Component, if not null.

For the sake of clarity, the Conceptual UI will be referred to as the CUI in the remaining of this document.

3.3 Conceptual Components

Any platform user-interface in order to be effective only needs the three following basic element types:

1. Containers that are sets of elements. Some elements within a container may be containers as well, allowing a hierarchical organization of the elements required to build a user-interface.
2. Controls where selections from the end-user can be made, allowing him to command the application logic.
3. Texts where end-user can enter character strings that are interpreted either by the application logic or by other end-users.

In conventional user interface APIs, a complex hierarchy of component types is declared to express those three UI basic types. In Graphical user-interfaces, windows, forms and menus are examples of containers; buttons, check boxes, list boxes, menu items are examples of controls; text fields, edit boxes, and note boxes are examples of texts. Our purpose in this invention is to mirror those three UI basic types into three and only three Conceptual Components. These Conceptual Components encapsulate all the required container, text and control functionality making it possible for the software developer to design a complete and effective conceptual user-interface. Consequently, a conceptual user-interface is fully portable across all platforms and devices. Furthermore, using this new UI modeling approach, the software developer will not have to wonder which specific UI components must be used for a specific target platform or device. He will just have to define which specific functionality is needed, translate it into a conceptual user-interface with Conceptual Components and the choice of the required specific native components will be automatically made for him in the rendering process.

So, a Conceptual Component is the base unit of a conceptual user-interface that can become either a container, a control or a text. For the sake of clarity, a Conceptual Component will be referred to as a CC in the remaining of this document. CCs include Container CCs, Control CCs and Text CCs. These are the three CC types.

An abstraction model such as the one enclosed in this invention is used to organize in a user-friendly way many values that will be presented and modified by the end-user. To be efficient for the rendering on various client devices, a user-interface abstraction model needs to distinguish between homogeneity and heterogeneity for its values: which ones have the same attributes (e.g. a two-dimensioned array of text cells in a spreadsheet application) and which ones don't (e.g. a set of controls and texts). Thus a CC should be able to hold multiple values if necessary, to implement the concept of homogeneity (for example, in Windows a list box holds a 1-dimensioned array of states). For this reason,

each CC has a matrix of values. The value type depends on the CC type. The value type of a Container CC is a set of other CCs (and some of them can also be Container CCs, which makes up a CC tree). The value type of a Text CC is an editable text. The value type of a Control CC is a selected state.

Each value within the CC is also associated to an image. So a value is composed of a CC set, a text or a selected state and an image. If the image width and height are greater than zero in the application logic, the image will be used in the rendering when possible (mostly in GUIs), since not all user interfaces can display graphics.

Referring to FIG. 3, shown here is the inheritance diagram representing the three CCs needed to build conceptual UIs. The Container CC **60**, the Control CC **70** and the Text CC **80** are all inherited from the CC **50**. The CC **50** contains the common functionality of Container, Control and Text CCs. And since any CC value can be represented by an image, the image processing resides in the CC **50** base class. So, the three CC types are sufficient to define any user interface.

Since a CC holds a matrix and since a matrix can spread on many dimensions (multi-dimensioned matrix), four mathematical concepts are to be understood before getting to the inner working of a CC.

The concept of a d-number ("d-" stands for "dimensioned-") is a multi-dimensioned number, composed of a number of dimensions, greater than or equal to 0 (can be more than 3 since it is an abstract model), and a set of integer values associated to each dimension. For example, {3, 2, 5} is a 3d-number with values 3 for the first dimension, 2 for the second and 5 for the third dimension. The 0d-number is expressed as {}. The number of dimensions and the values of a d-number may be modified. Operators such as =, +, -, <, >, <=, >=, ==, !=, may apply between two d-numbers. When these operators are used, it is assumed that both d-numbers have the same number of dimensions and that the operator is applied for the corresponding values of the two operands for all dimensions. For example, {3, 4} < {6, 5} is "true" because 3 < 6 and 4 < 5, but {1, 3} < {2, 2} is "false", since 1 < 2 but 3 > 2 (the operator < is "false" for the second dimension). When one operand is 0, the 0 in fact means a d-number that has a number of dimensions equal to that of the other operand and that all values are 0. So {6, 5} >= 0 is "true" but {6, -3, 2} > 0 is "false". There are many modification methods, like in FIG. 4, that used type dnum for parameters, which is the type of a d-number. Value dnumnull always refers to the 0d-number {}.

The concept of d-size is a d-number used to represent a multi-dimensioned area. The number of units in the area is the product of all dimension values. For example, d-size {7, 6} describes a 2-dimensioned area of 7 x 6 = 42 units; a d-size {7, 6, 3} describes a 3-dimensioned area of 7 x 6 x 3 = 126 units. The 0d-size {} always has 1 unit. The units within the d-size are undefined.

The concept of a d-coordinate is a d-number used to locate a unit within a d-size. The coordinate must always have the same number of dimensions than its corresponding d-size. Usually, d-coordinate >= 0 and d-coordinate < d-size (or d-coordinate <= d-size for element insertion purposes). For example in a d-size {7, 6, 3}, a d-coordinate should be

3-dimensioned and can vary between a minimum of {0, 0, 0} and a maximum of {6, 5, 2}. The 0d-coordinate unit always refers to the only unit of 0d-size { }.

The concept of a matrix is a collection of values of the same type, located into the units of a multi-dimensioned area. The matrix always has a d-size and content, which is the set of elements. Elements of the matrix can be of any data or object type (number, structure, array, even another matrix).

Referring to FIG. 6 and for the rest of this document, we employ the term "Values Matrix" to mean either a Matrix of Containers **360** if the CC is a container, a Matrix of Selections **400** if the CC is a control or a Matrix of Texts **450** if the CC is a text. The Instances Matrix **360**, Selections Matrix **400** and Texts Matrix **450** will be detailed in further sections of this document. Note that the CC type cannot be modified throughout his lifetime.

With reference to FIG. 4, the CC **50**, following the object-oriented concept, contains Data **105** and Modification Methods **110**. The Data **105** of the CC encapsulates three matrices that, along with the Values Matrix, will form the complete user information: the Labels Matrix **115**, the Read-Only Flags Matrix **120** and the Images Matrix **125**. The Modification Methods **110** are methods used to modify or update the Data **105** of the CC **50**. In the Modification Methods **110** column, Initialization time always means that the associated Data **105** member is set once at the creation of the CC. All information set at Initialization time cannot in any way be modified for the CC lifetime.

Still referring to FIG. 4, the Base d-size **150**, entered at Initialization time will be used as the foundation for the calculation of the Labels Matrix **115** d-size, Read-Only Flags Matrix **120** d-size, Images Matrix **125** d-size and also the Values Matrix d-size. Referring to FIG. 8, it is clearly shown that the Base d-size **150** is used to calculate these four different d-sizes that are used by the CC. The calculation of those d-sizes from the Base d-size **150** is explained in further sections.

Returning to FIG. 4, the Constant Flags **175** is a list of flags used to put constraints on the Base d-size **150**. The number of elements in this list of flags must be equal to the number of dimensions of Base d-size **150** to have one Constant Flag **175** per Base d-size **150** dimension. When one Constant Flag **175** is "true", it means the corresponding Base d-size **150** value, specified at CC Initialization time, cannot be modified afterwards. When it is "false", the Base d-size **150** value, still set first at Initialization time, can grow or shrink via insertion or removal of elements. The Variable d-size is a d-size that includes only the dimensions of the Base d-size **150** where the Constant Flags **175** is "false"; this means the Variable d-size can have a number of dimensions less or equal than the Base d-size **150**. For example, consider a Base d-size **150** of {2, 3} and Constant Flags **175** equal to {true, false}, which means the first dimension value of the Base d-size **150** (2) is constant (constant flag is "true" on the first dimension), however the second dimension value of the Base d-size **150** (3) can grow or shrink (constant flag is "false" on the second dimension). The Variable d-size in this case includes only the second dimension where Constant Flag **175** is "false", resulting in a 1d-size of {3}. The *insert(dnum start, dnum range)* **155** (start >= 0, start <= Variable d-size and range >= 0) and *remove(dnum start, dnum range)* **160** (start + range <= Variable d-size **150**, start >= 0 and range >=0) will

insert and remove elements in the dimensions where Constant Flags **175** is "false". Notice that the start and range parameters in insert and remove methods should have the same number of dimensions than Variable d-size, not Base d-size **150**.

Referring now to FIG. 9, let's present the last example schematically. Consider now the following insert({2}, {1}). As presented in FIG. 9, the insertion took place only on the second dimension where Constant Flag is "false". After the insertion, the new Base d-size will be {2, 4}. Two new empty values were added to the matrix while the six existing values (V1 to V6) stayed in the new matrix. However V5 and V6 were shifted one row down because of the insertion position {2}. Consider now a remove({2}, {1}) method. The removal as presented will also take place on the second dimension changing the Base d-size to {2, 2}. Only the first four values (V1 to V4) stayed in the new matrix. Values V5 and V6 have been removed because of the removal position {2} matching their row.

Referring to FIG. 4, the Constant Flags **175** are also used for the rendering. A practical illustration of the use of the Constant Flags **175** in the rendering is the scroll bars existing in Windows. For example, when the number of elements displayed in a window is constant on the horizontal direction (Constant Flag **175** is "true" on the first dimension) there is no need for a horizontal scroll bar since all the existing elements are already displayed. However if the number of elements can grow (Constant Flag **175** is "false" on the first dimension), there must be a horizontal scroll bar that enables the end-user to scroll all over the elements. The same concept applies on the Vertical direction in Windows (considered in this example the second dimension). This shows that the modification of the Constant Flags **175** over any dimension can result into a clear modification in the final rendering of the client device.

The End-User Resize Flags **180** are, just like the Constant Flags **175**, a list of flags, which number of elements is also equal to the number of dimensions of Base d-size **150** (to have one End-User Resize Flag **180** per Base d-size **150** dimension). A Resize Flag **180** with value "true" means that the end-user is allowed to change the corresponding Base d-size **150** value by sending insert/remove events to the CC. A "false" flag means the end-user cannot insert/remove units from the corresponding Base d-size **150** value. Therefore, the user is forbidden to send insert/remove events. Notice that when the Constant Flags **175** is "true" on a dimension, the End-User Resize Flag **180** is automatically set to "false" on this dimension. The user cannot change the size of a dimension when it is already set constant. However if the Constant Flag **175** is "false", the End-User Resize flag **180** can be "true" or "false", allowing or prohibiting the end-user to change the dimension size. Referring to FIG. 13, a practical illustration of the use of the End-User Resize Flags **180** is a multiple document interface (MDI) in Windows. In the conceptual UI model, a MDI window is presented by a CC of 1-dimensioned matrix, where the Constant Flag is set to {false}. In a MDI window, any of the documents (document1 **1050**, document2 **1055**, document3 **1060**) is subject to insertion/removal. In this example the end-user is allowed to close an existing document and create a new document (End-User Resize Flag is "true").

Referring to FIG. 4, the Directions **185** data member is a list of directions, which number of elements is also equal to the number of dimensions of Base d-size **150** (to have one

Direction **185** per Base d-size **150** dimension). Each Direction **185** in the list takes one of the three possible values: Width, Height or Depth. It specifies the direction of its corresponding Base d-size **150** dimension that will preclude when the CC is rendered to the user. Any rendering must occur in a 3-dimensioned space, which forces conversion back and forth between abstract n-dimensioned CCs and its corresponding concrete, rendered 3-dimensioned space (those n-dimensioned to 3-dimensioned and 3-dimensioned to n-dimensioned conversions will be explained later in the rendering part). The Directions **185** are used in the rendering for this conversion process.

Referring to FIG. 4, the Default Action Flag **190**, if set to "true", specifies that no event generated from user interaction in the CC will be passed on to the application logic, for further processing. Instead, the default behavior of user interaction in the CC will take place, without notifying the application that a change has occurred. For example, for most text-entry fields, the application never changes its normal behavior based on modification events such as additions and removals of letters, and therefore the application should not know about these events. The Default Action Flag **190**, if set to "true", can considerably speed up processing when there is a long delay between the user interaction on the client device and the reception of the event by the application. If the Default Action Flag **190** is set to false, user input is temporarily disabled when the user modifies the component. The user input will be re-enabled when the *Push(CC)* **65** (the CC may be null) and then *Pop()* **70** methods in FIG. 2 are called. In the meantime, the user is prohibited from entering other information while a previous command is still executing. For example in an address book, after the user has entered the first name, surname and phone of a contact he presses a submit button, he must be prevented from typing other information while the submission is being processed. Any attempt to modify the application at this time will be discarded, ensuring a safe use of the application.

Referring to FIG. 4, the Semantic Type **195** is a string that is used to identify a convention if applicable. Many applications have user interface fields that refer to known conventions, such as date, time, phone number, postal code, password, etc. Many user interfaces have specific controls (semantic controls) that handle these so commonly user fields. Their look & feel might be different across platforms, but ultimately they refer to the same human convention. If the CC refers to a human convention, the Semantic Type **195** string will be one entry from a pool of predefined convention strings like "date", "time", "zip code", and "password". If the user interface does not have a semantic control for a given convention, the Semantic Type **195** string is not used. An empty string in the Semantic Type **195** indicates that the CC does not refer to any human convention.

The User-Defined Flags **200** respectively address the Labels Matrix **115**, the Read-Only Flags Matrix **120** and Values Matrix (including the Images Matrix **125**). Although those three matrices are detailed later, consider them as matrices of user information elements for now. By default, these information elements will be stored in internal memory by the Rendered UI, on behalf of the programmer. However, if a User-Defined Flag **200** is "true", its corresponding information elements must instead be defined by the programmer in his source code, in fact the application logic (only the viewed part of the information will reside in the Rendered UI memory, for displaying purpose). The programmer will have to redefine the retrieval methods to get accurate data from its

proprietary storage mechanism. The User-Defined Flags **200** are usually required when the information consumes too much memory to be blindly stored by the Rendered UI internal mechanism. For example, in an Excel sheet, the cells are not all stored in Windows Rendered UI memory using a huge string array. In this case, the Excel programmer holds a structure where only the non-empty cells are stored. The programmer of a conceptual Excel application must set the sheet component User-Defined Flag **200** for the Values Matrix to "true" and establish links between the sheet component and his own structure by redefining the retrieval methods. The User-Defined Flags **200** are also useful when all the corresponding information elements can be retrieved using a mathematical formula. Instead of storing all information elements of the formula in memory, by setting to "true" the User-Defined Flags **200** the programmer can dynamically call the formula within the retrieval method when needed and avoid wasting memory space.

3.3.1 Labels Matrix

Referring to FIG. 4, the Labels Matrix **115** is used to describe the Values Matrix to the user. A label is a small description that is composed of a text string and a bitmap. Referring to FIG. 8, the Labels Matrix d-size **755** is equivalent to the Base d-size **150**.

Referring to FIG. 4, the String Max Length **215** contains the maximal length the text string of any label in the Labels Matrix **115**. If the String Max Length **215** if 0, no text string can be entered within labels.

The Bitmap Size **220** specifies the size, in pixels, of all bitmaps in Labels Matrix **115**. If the Bitmap Size **220** width and/or height are equal to zero, then the Labels Matrix **115** cannot contain any bitmap.

Notice that if in the Labels Matrix **115**, both String Max Length **215** and Bitmap Size **220** are zeroed, the Labels Matrix **115** cannot contain any label is therefore not used to qualify the Values Matrix.

The Labels Matrix **115** content will be either stored in the Buffered Storage System **225** residing in the application logic, the Default Storage System **230** residing in the CC or a User-Defined Storage System **245** residing in the application logic, depending on the label User-Defined Flag **200** and the Constant Flags **175**.

If the label User-Defined Flag **200** is set to "false" and for all Base d-size **150** dimensions, the Constant Flag **175** is "true" (constant), the Buffered Storage System **225** is used. In this case, the application must supply at initialization time a buffer of length (String Max Length **215** + 1) x Product of Labels Matrix **115** d-size that contains the text strings if String Max Length **215** is greater than 0 and, if Bitmap Size **220** is not null, a buffer of length Size Of Bitmap Structure x Product of Labels Matrix **115** d-size that will contain bitmaps.

If the label User-Defined Flag **200** is set to "false" and at least for one Base d-size **150** dimension a Constant Flag **175** is "false" (not constant), the Default Storage System **230** is used. Empty label elements can be created with *insert(dnum start, dnum range)* **155**.

For both Buffered Storage System **225** and Default Storage System **230**, the method *setlabel(char *label, dnum index = dnumnull)* **235** is used to modify the text string of the label element pointed to by index d-coordinate and method *setbitmap(Bitmap *bitmap, dnum index = dnumnull)* **240** must be used to modify the bitmap of the label element pointed to by index. In both methods, the index parameter is a d-coordinate within Labels Matrix **115** d-size (index >= 0 and index < Labels Matrix **115** d-size).

If the label User-Defined Flag **200** is set to "true", the User-Defined Storage System **245** is used. The *updatelabels(dnum pos, dnum range)* **250** can be used to refresh the label rendering whenever a corresponding change in the proprietary label storage mechanism of the application logic has occurred. The pos and range parameters must have the same number of dimensions than Labels Matrix **115** d-size, pos >= 0, range >= 0 and pos + range <= Labels Matrix **115** d-size.

3.3.2 Read-Only Flags Matrix

Referring to FIG. 4, the Read-Only Flags Matrix **120** content determines whether or not the Values Matrix elements enable user input. If "true" user input is disabled, otherwise it is enabled. Referring to FIG. 8, the Read-Only Flags Matrix d-size **785** is equivalent to the Values Matrix d-size **780**. The calculation of the Values Matrix d-size **780** is explained later in this document.

Returning to FIG. 4, the Constant Flag **270** is set at initialization time. If its value is "true", all Values Matrix elements in the CC are considered read-only for the CC lifetime. Consequently, the Read-Only Flags Matrix **120** will never be created nor accessed by the application logic. Methods *updateroflags(dnum pos, dnum range)* **290** and *setroflag(boolean flag, dnum index = dnumnull)* **280** cannot be called. However, if the Constant Flag **270** is set to "false", the Read-Only Flags Matrix **120** values can be set to "true" or "false" to forbid or allow user editing on the corresponding Values Matrix elements within the CC. Methods *updateroflags(dnum pos, dnum range)* **290** and *setroflag(boolean flag, dnum index = dnumnull)* **280** can be called to modify Read-Only Flags Matrix **120** content.

The Read-Only Flags Matrix **120** content will either be the Default Storage System **275** residing in the Rendered UI or a User-Defined Storage System **285** residing in the application logic, depending on the read-only User-Defined Flag **200**.

If the read-only User-Defined Flag **200** is set to "false", the Default Storage System **275** is used. The method *setroflag(boolean flag, dnum index = dnumnull)* **280** modifies the read-only flag element pointed to by index. The index parameter is a d-coordinate within Read-Only Flags Matrix **120** d-size (index >= 0 and index < Read-Only Flags Matrix **120** d-size).

If the read-only User-Defined Flag **200** is set to "true", the User-Defined Storage System **275** is used. Method *updateroflags(dnum pos, dnum range)* **290** will refresh the read-only flag rendering whenever a corresponding change in the proprietary read-only storage mechanism of the application logic has occurred. The pos and range parameters must

have a number of dimensions equal to the Read-Only Flags Matrix **120** d-size, pos >= 0, range >= 0, pos + range <= Read-Only Flags Matrix **120** d-size.

3.3.3 Images Matrix

The Images Matrix **125** is used by the application logic to express CC values (either containers, control selections or texts) with a graphical representation. Referring to FIG. 8, the Images Matrix d-size **790** is equivalent to the Values Matrix d-size **780**. The calculation of the Values Matrix d-size **780** is explained later in this document.

Returning to FIG. 4, in any user interface, an image is a two-dimensioned array of pixels. The 2D size **320** specifies, in pixels, the width and height of all images in the Images Matrix **125**. It is set at Initialization time. Both width and height must be greater than or equal to 0. If 2D size **320** is not 0 for width and height, and the values User-Defined Flag **200** is set to "false", the images are stored in a Default Storage System **325** and can be modified via a set of *Text & Graphical Commands* **330**. The index parameter at the end of all *Text & Graphical Commands* **330** is a d-coordinate within Images Matrix **125** d-size (index >= 0 and index < Images Matrix **125** d-size). If 2D size **320** is not 0 for width and height, and the values User-Defined Flag **200** is set to "true", the images are stored in a User-Defined Storage System **335**. Method *updatevalues(dnum pos, dnum range)* **340** (pos >= 0, range >= 0, pos + range <= Images Matrix **125** d-size) will refresh the image rendering whenever a corresponding change in the proprietary value storage mechanism of the application logic has occurred. If 2D size **320** width or height is 0, the Images Matrix **125** becomes inactive and is not used by the application logic.

The application logic may use the Images Matrix **125** to precisely control the rendering and user input of each Values Matrix element (Children Set instance, selection or text as explained in FIG. 6). The rendering process of the Values Matrix is decided as follows.

- If the Rendered UI targets a sophisticated platform where the user can view/manipulate images and the Images Matrix **125** is used (that is, the 2D size **320** applicable to all images is not null), the Images Matrix **125** takes precedence over the Containers, Selections or Texts Matrix and images are displayed. Images from the descendent CCs (if the CC is a container) are also displayed within the CC images, at the specified position.
- If the Rendered UI targets a limited platform where the user can barely view/manipulate images or the Images Matrix **125** is not used (that is, the 2D size **320** applicable to all images is null), the Values Matrix is directly used by the Rendered UI, which will use device components to render the values.

3.3.4 Values Matrix

Before detailing the Values Matrix content, which is dependent of the CC type, consider the Values Matrix d-size, which calculation method applies to all CCs.

The Values Matrix d-size is calculated from two factors: d-size accumulation and Container d-size.

D-size accumulation, expressed with operator |, happens when each element of a matrix is another matrix, as it happens in CC trees. For example, a two-dimensioned matrix of d-size {4, 2} may contain elements that are three-dimensioned matrices of d-size {3, 5, 2}. The d-sizes of the two matrices can be accumulated to form one matrix of d-size {4, 2, 3, 5, 2} (e.g. {4, 2} | {3, 5, 2} = {4, 2, 3, 5, 2}). D-size accumulation is associative (e.g. {4, 2} | {3, 5, 2} is equivalent to {4, 2, 3} | {5, 2}), but not commutative (e.g. {4, 2} | {3, 5, 2} is not equal to {3, 5, 2} | {4, 2}).

Referring to FIG. 8, the Container d-size **770** of a CC is itself an accumulation of all the Base d-sizes **150** of its parent Container CCs, from the root Container CC to the Container CC that directly references the CC. The Container d-size **770**, to exist, must apply to a linked CC, else it is always {}. With reference to FIG. 7A that shows a linked CC tree with the base d-size of each CC (the Container CC **60c** has no children). The Root Container CC **60a** will always have Container d-size {} (1 unit). The CC **50d** and Container CC **60b** Container d-size is {2, 4} (8 units), since they are the children of Root Container CC **60a**. The CC **50e** and CC **60c** Container d-size is {2, 4, 5} (40 units). This Container d-size is the accumulation of CC **60a** and Container CC **60b** base d-sizes.

Referring to FIG. 8, for any Container CC, the Values Matrix d-size **780** is the d-size Accumulation **775** of its Container d-size **770** and its own Base d-size **150**. For example, returning to FIG. 7A, the Values Matrix d-size of Container CC **60c** would be {2, 4, 5} | {4, 3, 2} = {2, 4, 5, 4, 3, 2} (960 units).

The time has come to detail the Values Matrix content for all three CC types.

3.3.4.1 Container CC

Referring to FIG. 5A, a Container CC **60** is used as a set of other CCs, referred to as child CCs. A Container CC **60** is used to build complex user interfaces, by building a tree of CCs. When rendered, a Container CC **60** may end up, for example, as a menu or a window containing controls built from child CCs.

The Children Set **365** is a list of references to child CCs, they must be set immediately at initialization time. The Children Set **365** can be empty if desired. The Children Set **365** is subject to restrictions if the Semantic Type is not null and refers to a valid convention. In this case, the Children Set **365** (and all the descendants) must follow that convention to be effective. Furthermore, the Read-Only Flags Matrix values and the user-defined flag for values of the Container CC **60** also apply for all the child CCs values. For example, if a Container CC **60** is not editable (read-only flag is "true") its entire child CCs are not editable as well.

Each child CC also contains an internal Parent variable that will always reference its current Container CC **60**. When a CC is not referenced by a Container CC **60**, the Parent variable is set to null. Only CCs where Parent is null can be pushed in and popped out of the CUI. The Root Container CC **60** of each tree is the Container CC **60** that is directly referenced by the CUI.

Since the Children Set **365** of the Container CC **60** is set at Initialization time, the order of creation of the CCs that will be part of a CC tree becomes fundamental. CCs must be

created following their bottom-up order in the tree. For example, referring to FIG. 7A, a convenient creation order of the CC tree is by first creating CC {7, 3} **50d**, then CC { } **50e**, then Container CC {4, 3, 2} **60c**, then Container CC {5} **60b** that immediately references CC { } **50e** and Container CC {4, 3, 2} **60c** at initialization time, and finally Root Container CC {2, 4} **60a**, which references CC {7, 3} **50d** and Container CC {5} **60b** at Initialization time. The destruction order of the CCs in the CC tree always happens in the reverse order of their creation.

Referring to FIG. 5A, all CCs referenced by a linked Container CC **60** are linked as well and will be rendered to the user. It means that a root Container CC **60** that is pushed in the CUI is not alone to become linked. All its descendants are also linked and consequently rendered to the user. The entire tree is considered linked.

The main difference between Container CCs **60** and other CC types is that the Values Matrix unit is not predefined (like selections for Control CCs and texts for Text CCs). The Container CC **60** uses the Children Set **365** to define the unit of its Values Matrix. The Children Set **365** is seen as a structure, an organization, a type, a template or a model that cannot be modified throughout the Container CC **60** lifetime. Instances of this Children Set **365** structure will populate the Values Matrix. Referring to FIG. 7B, the Container CC **60d** has two children, namely CC A **50f** and CC B **50g**, that could be of any CC type. The Values Matrix, which d-size is {3}, will have instances of the set composed of CC A **50f** and CC B **50g**. Since the base d-size of CC A **50f** is {2}, each unit in the Container CC **60d** Values Matrix will have two instances of CC A **50f**. Since the base d-size of CC B **50g** is {3}, each unit in the Container CC **60d** Values Matrix will have three instances of CC B **50g**. The rendered UI (RUI **750**) shows the layout of the output rendered to the end-user and the coordinates of CC A **50f** and CC B **50g** values.

Referring to FIG. 5A, the Children Direction **370**, set at initialization time, specifies the direction on which all the child CCs will be represented, when rendered to the user. The Children Direction **370** takes one of the three possible values: Width, Height or Depth, always considering that the rendering is occurring in a 3-dimensioned space. The Children Direction **370** gives the user the ability to choose the space direction where child CCs accumulate. It is strictly applied when the Container CC has dimensions in its Values Matrix. However, when the Container CC has no dimension in its Values Matrix (its Children Set **365** is displayed once and is never replicated), the Children Direction **370** is superseded by the device proprietary field positioning rules. In this case, the children are positioned according to the client device conventions. On certain devices though, the Children Direction **370** might still influence the rendering, which is why it is important to define it even when there is no dimension in the Values Matrix. For example, consider FIG. 12A. The tax credit form **1000** has a Children Direction of Width or Height, which means all fields (Name **1005**, Address **1010** and Age **1015**) are displayed on the same panel. Referring to FIG. 12B where the same Container CC has a Children Direction of value Depth, the rendering corresponds to a Tab Control **1025** where each tab item (Name **1030**, Address **1035** and Age **1040**) refers to one piece of information (one child CC) and the end-user can switch between tab items. Referring to FIG. 7B, the Container CC **60d**

and the CC A **50f** have one dimension along the width, whereas the CC B **50g** has one dimension along the height, as shown in the rendered UI (RUI) **750**.

Returning to FIG. 5A, the Select CC Flag **375**, set at Initialization time, enables the application logic to force the end-user to use only one child at a time. When true, the end-user can only interact with the child CC indexed by Selected CC **378**. The application logic can modify the Selected CC **378** with method *selectcc(int childindex)* **385**, allowing the end-user to interact with another child. If the childindex parameter is less than 0 or greater than the number of child CCs in the Children Set **365**, no child CC is presented to the end-user to interact with.

Returning to FIG. 5A, the Instances Matrix **360**, which is the Container CC **60** Values Matrix, holds the instances of the Children Set **365**. Method *updatevalues(dnum pos, dnum range)* **395** (pos >= 0, range >= 0, pos + range <= Instances Matrix d-size) will refresh the rendering whenever a corresponding change in the Children Set **365** instances has occurred.

3.3.4.2 Control CC

Referring to FIG. 5B, a Control CC **70** is composed of a States Matrix **405** containing the states and a Selections Matrix **400** (which is the Control CC **70** Values Matrix) storing the different selection d-coordinates of States Matrix **405** units. These selections are available for the end-user to control the application logic behavior. The following example clarifies the difference between a State and a Selection. Referring to FIG. 10, a Control CC **70** is presented with all its Data Members **105** and their actual Value **505**. As shown, the Control CC **70** has a Base d-size **150** of {2} and "me:" "you:" as label strings. The States Matrix of the Control CC **70** has a d-size of {4} and "A", "B", "C", "D" as label strings. Given the Control CC **70** Data Members **105** values, for the Windows platform, within a window client area, the rendering of the Control CC **70** comprises two combo boxes having both the same States Matrix (4 states). The rendering rules that were used in this example will be explained in a later section of this document. In this example, the end-user current combo box selections are "B" and "D". Accordingly, the Selections Matrix will be composed of two d-coordinates indicating the right states indexes, namely {1} and {3}. This example shows that the States Matrix and the Selections Matrix are different but complementary for building complete and complex controls.

Referring to FIG. 5B, the States Matrix **405**, as for the CC **50** from FIG. 4, is constituted from:

1. States Matrix d-size, like the Base d-size is the d-size of the States Matrix **405**.
2. A set of attributes equivalent to those of the CC matrix, applied to each States Matrix **405** dimension, including: Constant Flags, End-User Resize Flags and Directions. Those attributes have been explicitly defined for the CC matrix.
3. A Labels Matrix similar to the one defined in the CC.

Referring to FIG. 11 is shown, for a Control CC, the interaction between the States Matrix d-size **760**, the Bitmap Size **220** and the String Max Length **215** when it comes to

rendering the CC. Note that for each combination of States Matrix d-size **760**, Bitmap Size **220** and String Max Length **215**, the rendering shown takes the three values into account, but can be different from one platform to another (it is shown with Windows in the example) and also from one context to another. Within a menu context for example, menu items would be chosen for the rendering. The states are composed, when String Max Length **215** is > 0, from labels extracted from the states Labels Matrix. For States Matrix dimensions where String Max Length **215** is zeroed, the States are unnamed.

Referring to FIG. 11, consider first a States Matrix d-size **760** of { } (first row) and a null Bitmap Size **220** (first column). The generated rendering represents a selection over { } state (1 state since { } always represent 1 element). The selection can change from nothing ({ }) to nothing ({ }), which why is it is expressed in Windows as a button. Depending on the String Max Length **215**, the rendered button can contain a label (String Max Length **215** is > 0) or be represented as an empty button (String Max Length **215** is 0). The current selection of a button is always { }.

Still referring to FIG. 11, consider now a States Matrix d-size **760** equal to {4} (second row) and a null Bitmap Size **220** (first column). The generated rendering must represent a selection over {4} (4 states). Those states can be either unnamed (String Max Length **215** is 0) or named (String Max Length **215** is > 0). If states are unnamed, it can be expressed in Windows as a track bar with 4 different states (since in Windows track bars do not contain any labels). However, if the states are named, a possible Windows rendering could be a combo box having 4 states (combo box can be chosen because it is a Windows component that contains labels). So for the same States Matrix **760** d-size value, rendering possibilities change according to the String Max Length **215** values. Notice that as presented in FIG. 11, in both renderings the current selection is {1}.

Consider now a States Matrix d-size **760** equal to {2, 3} (third row) and a null Bitmap Size **220** (first column). The generated rendering must represent a selection over {2, 3} (6 states). Those states can be either unnamed (String Max Length **215** is 0) or named (String Max Length **215** is > 0). If states are unnamed, it can be expressed in Windows as a check box that sets the first dimension value and a track bar that sets the second dimension value. Both the check box and the track bar were chosen for rendering because they do not contain labels. However, if the states are named, it can be expressed in Windows as radio boxes setting the first and second dimension value. The radio boxes are spread in width and height since the State Matrix d-size **760** is two-dimensioned. As presented in FIG. 11, in both renderings the current selection is {1, 1}.

Notice that all the cases discussed, Bitmap Size **220** was null (there is no bitmap). In the case Bitmap Size **220** was not null, the rendered components would have had a bitmap associated to a string (if String Max Length 215 is > 0) or only a bitmap (if String Max Length is 0). A rendering possibility is also shown for each case. Please notice also that the choice of different Windows components is not discussed in the samples shown in FIG. 11. The rendering rules will be discussed in detail in the last part of this document.

Returning to FIG. 5B, The Selections Matrix **400** (the Control CC **70** Values Matrix) holds all the selections in the Control CC **70**. If the values (d-coordinates in this case) User-Defined Flag is set to "false", the Default Storage System **420** will store the

Selections Matrix **400** of the Control CC **70**. The selection values can be changed anytime with method *setselection(dnum sel, dnum index = dnumnull)* **425** (sel >= 0, sel < States Matrix **405** d-size, index >= 0 and index < Selections Matrix **400** d-size). The end-user may also send setselection events to the application logic, with the same parameters than *setselection(dnum state, dnum index = dnumnull)* **425**.

If the values User-Defined Flag is set to "true", the Selections Matrix **400** is stored in a User-Defined Storage System **430**. Method *updatevalues(dnum pos, dnum range)* **435** (pos >= 0, range >= 0, pos + range <= Selections Matrix **400** d-size) will refresh the state rendering whenever a corresponding change in the proprietary state storage mechanism of the application logic has occurred.

3.3.4.3 Text CC

With reference to FIG. 5C, a Text CC **80** is used to enter text strings. The Text CC **80** consists of a collection of characters such as 'U', 'S' or '$'. There are also control characters, such as LF (line feed) to indicate a line is over and FF (form feed) to indicate a page is over.

The Number of Dimensions **455**, entered at Initialization time, specifies the number of dimensions of the text, which is greater or equal to 0. The Text CC **80** is multi-dimensioned, as a collection of characters can span over multiple dimensions. A 0d text is always a single regular character (not a control character). No extra characters can be added or removed. A 1d text is one list of characters (a line), whereas a 2d text is a list of list of characters (a page), and so on. On any dimension, insertion and deletion are possible. Control characters are used to switch to the next unit in a given dimension. LF switches to the next unit in dimension 1 (a switch to the next line) and FF switches to the next unit in dimension 2 (a switch to the next page).

The Maximal Length **460**, set at Initialization time, is a number indicates the maximal character count for all texts in the Values Matrix. The number of characters in any Text CC **80** text value will never exceed the Maximal Length **460**. However, if the Maximal Length **460** is set to -1, the text can have any length and the number of characters in the Text CC **80** can grow indefinitely.

The Text View **465**, entered at Initialization time, is a d-size specifying the number of characters rendered to the user on each text dimension. It can be considered as a view area, expressed in text character units. The Text View **465** must have the same number of dimensions than the text Number of Dimensions **455**. It is used in the rendering process to define the dimensions of the displayed text component, especially when the Values (texts) Matrix has multiple dimensions.

The Texts Matrix **450** (the Text CC **80** Values Matrix) holds all texts in the Text CC **80**. If the values (texts in this case) User-Defined Flag is set to "false", the Default Storage System **470** contains all texts of the Texts Matrix **450**. The *settext(dnum pos, long lengthtoremove, char *string, long stringlength, dnum index = dnumnull)* **475** method modifies, for the text pointed to by index, its content, at the d-coordinate specified by pos, by removing lengthtoremove characters and adding stringlength characters taken from

string. For this method, index >= 0, index < Texts Matrix **450** d-size, pos must point to a text character and from pos, lengthtoremove must not exceed the number of characters. There is also a settext event that can be invoked from the end-user. It has the same parameters as *settext(dnum pos, long lengthtoremove, char *string, long stringlength, dnum index = dnumnull)* **475**.

If the values User-Defined Flag is set to "true", the Texts Matrix **450** is stored in a User-Defined Storage System **480**. Method *updatevalues(dnum pos, dnum range)* **485** (pos >= 0, range >= 0, pos + range <= Texts Matrix **450** d-size) will refresh the text rendering whenever a corresponding change in the proprietary text storage mechanism of the application logic has occurred.

4. Rendered UI

The Rendered UI is the concrete, 3D replica of the CUI that is visible and controllable from the end-user. Like the CUI, it is also a stack of trees that are modeled according to their corresponding ones in the CUI, but containing Device Components instead of conceptual components. Unlike the three kinds of CCs, the set of possible Device Components is different from one client platform to another. For the sake of clarity, the Rendered UI will be referred to as RUI, for the rest of the document. Also, Device Components will be referred to as DCs.

A DC is a tangible version of a linked CC (a CC accessible from the CUI) that uses for rendering specific, native components that are proprietary to the device operating system such as, in Windows for example, overlapped window, push button, checkbox, dialog box, toolbar. Therefore, DCs are different from one platform to another. For the sake of clarity, the Native Component used by DCs will be referred to as NC for the rest of the document.

The RUI must fill the following tasks.

1. It selects the required DC types to render CCs. A DC, contrary to a CC, is not conceptual, is truly rendered to the end-user using NCs. Many types are possible. There is always one associated DC for one linked CC, but its type is selected depending on the parent DC type and the CC attributes set at initialization time.
2. When requested by the application logic, it takes device-specific events from the end-user and converts them into Conceptual Events sent to the corresponding CC, if applicable. All Conceptual Events are returned to and then processed by the application logic.
3. It stores all the CC values that are not user-defined. Retrieval and modification methods can be called to get and set values.

The number of DC types, unlike the three CC types, can be extended to reflect different client contexts. For instance, in Windows, a Control CC with a 0-dimensioned states matrix can be rendered as a Menu Command DC or a Push Button DC, depending on the context (overlapped window menu or client window).

The RUI makes the link between an abstract CUI and a specific platform user interface made of NCs. Therefore, its rules and links to device NCs must be coded once per platform by using platform-centric methods.

Any RUI, to render a CUI, must hold a structure that tracks the one in the CUI. As an example, referring to FIG. 20, schematically shown is a simple Calculator CUI **10** rendered with the Windows RUI **15** and the Output **1110** generated from the Windows RUI **15**. The Calculator CUI **10** has only one element pushed in the Stack **12**: a Container CC **60** that holds both the numeric display and the buttons matrix. The numeric display is a Text CC **80** with a 1-dimensioned text. The Control CC **70** contains the entire buttons matrix: its base d-size is {4, 4} (16 selections) among { } states (1 state). The application logic is responsible to set the matrix labels and to modify the numeric display when setselection conceptual events are received.

The Windows RUI **15** is, for a Windows environment a Modal window stack **1140** which is initially empty. When a CC tree is pushed in the Stack **12**, the Windows RUI **15** will also push a Windows DC tree on the Modal window stack **1140**. In the example, for each CC of the CC tree (Container CC **60**, Text CC **80** and Control CC **70**), there is a corresponding DC in the DC tree (Null DC **55**, Edit DC **1150** and Button DC **1155**). The number of DC types for the Windows environment is far larger than the three applicable for CCs to reflect the richness of the GUI.

The type of each DC in the tree is determined by examining the parent DC type and the CC attributes set at Initialization time. There is no parent DC type for the Container CC **60**, so by default for containers, a Null DC **55** is created. Since the Null DC **55** has no frame, it creates one (an overlapped window **1160** in this case because it is the Root DC and the first pushed DC in the RUI – it would have been a modal dialog box otherwise) and removes it when it is destroyed. This frame reference will be passed to all children of the Null DC **55**.

The Edit DC **1150**, associated to Text CC **80**, contains a matrix of Edit controls. Since the values matrix d-size is { }, there will be only one Edit control in the matrix. This Edit control will have as parent window the overlapped window **1160** sent by Null DC **55**.

The Button DC **1155**, associated to Control CC **70**, contains a matrix of Button controls with style Pushbutton. The values matrix is {4, 4}, allowing 16 selections within a state matrix of d-size { } (1 state). Within an overlapped window, the only child control allowing a state change from { } to { } is the pushbutton. When it is pressed, it triggers the setselection({ }, index within {4, 4}) event. Thus, there will be a matrix of 16 pushbuttons. These pushbuttons will have as parent window the overlapped window **1160** sent by Null DC **55**.

The resulting window **1165** in the Output **1110** column is laid out according to the DC order and the most natural layout of controls within the overlapped window. In this case, it is logical to place the button matrix below the edit field.

4.1 Proprietary Rendering

DCs act like intermediates between CCs and NCs. Any DC performs an implementation of a logical use case of NCs. For example, it is logical in Windows to create a dialog box, then child controls within the dialog box. It is not logical, even if it is technically possible, to create an edit child control within a push button child control. The set of DC types is created to ensure logical combinations of NCs, to manage logical usage of the targeted platform UI, when rendering a CC tree.

Referring to FIG. 16A, the steps required to render a CC tree in the CUI **10** stack begin by creating the DC tree in the RUI **15** stack. For each CC **50** in the CUI **10**, a DC **55** in the RUI **15** must be initiated with a correct DC type. Since there can be many types of DCs **55**, depending on the complexity of the platform user-interface, a set of DC decision flowcharts **1200** proprietary to the end-user device is required. Second, the DCs **55** in the RUI **15**, according to their type, must generate all NCs **75** that form the rendered output. The passage from a DC **55** to NCs **75** is based on a set of NC decision flowcharts **1250** proprietary to the end-user device. A set of DC decision flowcharts **1200** and a set of NC decision flowcharts **1250** for the Windows platform (in FIG. 18A to FIG. 18H and FIG. 19A to FIG. 18E) are provided in this document to provide a detailed example on how such flowcharts should be devised.

The creation of a DC tree in the RUI **15** follows the top-down method, as opposed to the CC tree in the CUI **10**, created by the bottom-up approach. It means the Root DC **55** is created first, then its children, then its grandchildren, etc., following the tree structure of the corresponding CC tree from the CUI **10**.

The CC-DC relationship is always one-to-one. Referring to FIG. 16B, two CCs, CC1 **50a** and CC2 **50b** are respectively associated to a DC1 **55a** and DC2 **55b** according to the DC decision flowcharts proprietary to the client device. However, the DC-NC relationship is always one-to-many. By using the proprietary NC decision flowcharts, the DC1 **55a** generates one NC, NC1 **75a**, whereas the DC2 **55b** generates two NCs, NC1 **75a** and NC2 **75b**. Furthermore, NCs can be reused by more than one DC types. For example, NC1 **75a** is used by both DC1 **55a** and DC2 **55b**.

Any set of DC types can be organized in a state machine to illustrate the creation order of DCs when generating DC trees. Referring to FIG. 17, the Windows state machine **1300** represents Windows DC types such as Tab Control **1365**, Menu Command **1430** and Overlapped Window **1325**. All states are terminal states, except for the states in italic (Start **1305**, Menu bar Creation **1400**, Status bar Creation **1450**, Control Creation **1350** and Submenu Creation **1500**) which are transitory states. The construction of a DC tree from a CC tree will follow this state machine and always begins at Start **1305**. Depending on the Root CC, a transition will be made either to Control Creation **1350** or Overlapped Window **1325**. If the transition is made to Control Creation **1350**, another transition is immediately required to determine the DC type of the Root CC, since Control Creation **1350** is transitory. This transition will end to one of many DC types, such as Combo Box **1375** or Unlabeled Control **1390**. Then, for children of the Root CC, other transitions begin from the current state. Thus, the state machine **1300** can be applied to all CCs of a

CC tree, allowing the construction of the DC tree. The state machine **1300** rules for all transition are detailed in FIG. 18A to FIG. 18H flowcharts.

Please note that more sophisticated state machines **1300** (with more states) can, and should, be devised for Windows. The one shown here just illustrates that construction of complex Windows rendering of applications can be performed from simple conceptual concepts. Also, the state machine (with accompanying flowcharts) techniques used in this case for Windows are equally applicable to any other platform, simple or complex and should be considered as guidelines for constructing rendering adapters.

Based on FIG. 17 state machine **1300**, now referring to all figures from FIG. 18A to FIG. 18H, a DC tree construction algorithm has been devised with a hierarchy of flowcharts, applicable to the Windows GUI. For the sake of simplicity, it does not take into account for all CCs the images matrix superseding the conceptual Values Matrix, the semantic type and for Container CCs, the select CC flag. Referring to FIG. 18A, the terminology used to understand the following flowcharts specifies that CCl0 means CC level 0 (Root CC in fact), CCl1 means CC level 1, which is any child of the Root CC, CCl2 means CC level 2, which is any grandchild of the Root CC, and so on. CCln means a CC at level n, and CCln+1 means any child of a CCln. The same leveling terminology also applies to DCs.

Referring to FIG. 18B, the main flowchart of the DC tree construction algorithm. At this point, we only determine whether the DCs will be within a modal dialog box purely made of child controls or within the main overlapped window, since many DC types such as Menu Selection DC and Status Line DC are only useable in overlapped windows (in dialog boxes, these DC types cannot be used). The rule of thumb for using overlapped window-related DC types is the CC tree being the first pushed on the CUI stack and the Root CC being a Container CC where children direction is depth (by convention for sophisticated GUIs with wide displays, the depth can always be processed in a proprietary fashion, to ease the access to information). In this case, the DC created from the CCl0 is the Overlapped Window DC itself, and children are assumed to be menu bar items for now. If not, the DC created from the CCl0 is a child control that will be enclosed within a modal dialog box.

Referring to FIG. 18C, the menu bar item construction flowchart verifies if the CC is qualified to become a menu bar item. In order to qualify, the CC must not be a Text CC, must be 0-dimensioned and constantly read-only. If the CC is a Control CC, its states matrix must either be 0-dimensioned (generating a Menu Command DC) or 1-dimensioned, along the depth, with labels (generating a Menu Selection DC). If the CC is a Container CC, there must be children aligned on the depth direction. In this case, a Submenu DC is generated. Any other kind of CC is assumed to build a corresponding DC which type is linked to the overlapped window status bar.

Referring to FIG. 18D, the status bar item construction flowchart verifies if the CC is qualified to become a status bar item. Basically, the CC must not be a Container CC, must be 0-dimensioned, not constantly read-only and having a string label. If the CC is a Control CC, the DC generated is a Status Control DC, where only the current selected state is displayed in the status bar, besides the label if any. If the CC is a Text CC, the text

number of dimensions must be less than or equal to 1 (a line or a single character). If so, a Status Line DC is generated. Any other kind of CC is assumed to build a corresponding DC which type is linked to the overlapped window client rectangle, where child controls take place.

Referring to FIG. 18E and FIG. 18F, the submenu item construction flowchart has almost the same behavior as the menu bar item construction flowchart from FIG. 18C. Referring to FIG. 18E, the main exception is to allow 1-dimensioned CCs to become submenu items, as long as the dimension constant flag is false, aligned on the depth direction and has labels. Furthermore, referring to FIG. 18F, there is another exception that happens when the states are unlabeled, constant and their number is 2 (generating a Menu Checkbox DC). Any other kind of CC is assumed to build a corresponding DC which type is linked to the overlapped window status bar in FIG. 18E or to the overlapped window client rectangle in FIG. 18F.

Referring to FIG. 18G, the child control construction flowchart verifies if the CC is qualified to become a child control CC. At this stage, depending on the CC attributes, the DC will be either Null DC, Tab Control DC, Edit DC, Push button DC, Combo Box DC, List Box DC, Radio Buttons DC or Unlabeled Control DC. Any other kind of CC cannot logically be constructed using child controls and is assumed to build a corresponding DC which type is generic. Generic DCs can render any CC and consequently are the DC typing bottom line that ensures the correct mapping of any CC tree for any platform, as explained in the next section.

Finally, referring to FIG. 18H, it is shown than a Generic DC can only contain other Generic DCs.

Referring to all figures from FIG. 19A to FIG. 19E, a set of flowcharts illustrates the creation of Windows NCs, starting from a created DC tree.

Referring to FIG. 19A, the NC creation flowchart for an Overlapped Window DC simply creates the overlapped window NC and calls the appropriate NC creation algorithm that applies to each DC child of the DC.

Referring to FIG. 19B, the flowcharts related to all menu and status bar DCs are illustrated. Correct menu items and submenu NCs are created for Menu Checkbox DCs, Menu Command DCs, Menu Selection DCs and Submenu DCs. Please note that NCs are created for all units of DC base d-size, in accordance with CUI rules regarding Values Matrix. Please also note that Windows toolbar items are created along with menu items when bitmapped labels are defined, to allow direct access to options. For the overlapped window status bar, the appropriate status bar parts NCs are created for Status Control and Status Line DCs. Since these DCs always have a 0-dimensioned Values Matrix, there is no need to loop through all base d-size units, which simplifies their NC creation flowcharts.

Referring to FIG. 19C and FIG. 19D, the NC creation flowcharts related to all child control DCs are displayed. Since all these DCs can have a Values Matrix that has dimensions, parsing through all Values Matrix units is required for creating NCs. Furthermore, for most DCs, the method createlabelNCs, detailed in FIG. 19E, has to be

called to create label-related NCs such as static controls, dialog boxes, ... The createNCs(Unlabeled Control DC) from FIG. 19D also shows that unlabeled controls have one anonymous selection child control per states matrix dimension, where the combination of child controls values form the selection (as shown in FIG. 11).

The positioning of child control NCs within dialog boxes can be done by using an algorithm that generates "lines" of NCs built from each DC. All possible combinations of lines can be compared based on two criteria: the amount of unused space saved on the dialog and the width/height ratio that must be closest to the Fibonacci golden ratio (~1.618 or 0.618), which defines the most harmonious proportion. The best combination will be used when rendering the dialog box. Other positioning algorithms can be used to lay out child controls within dialog boxes.

It has been demonstrated that for platform with sophisticated windowing interfaces, a DC tree can appear all at once to the end-user. For more limited platforms, in order to render complex DC trees within small displays, a navigation principle is often required. Referring to FIG. 21, a simple navigation example shows a Blackberry capable of displaying only one Children Set of a Container CC at a time. In order to navigate in the DC tree, consider the current end-user view **1600a**, shown in the Blackberry **1610a**, where the screen title has the Container CC **60a** label, fields are labeled from the Text CC **80a**, the Text CC **80b** and the Text CC **80c** and a menu option is taken from Container CC **60b**. The end-user, in order to navigate down to view **1600b**, must select in the Blackberry menu the Container CC **60b** label. When selected, the screen title label is extracted form Container CC **60b** and all fields and menu options are modified to reflect view **1600b** in the Blackberry **1610b**. The Blackberry menu will always have an extra "Close" option in order to go back from view **1600b** to view **1600a**. Besides this navigation principle, all principles and techniques that have been used in the Windows example, namely the state machine composed of DCs, the DC tree construction algorithm and the NC construction algorithms (all algorithms illustrated using flowcharts in this document) are equally applicable to the Blackberry and to any other platform.

4.2 Generic rendering

The first task to build the RUI is to transform an abstract n-dimensional space such as a CUI into a physical three-dimensional space that complies with the spatial nature of the universe. Even though most rendering devices use two-dimensional, delimited physical displays, at the logical level, the display is truly 3D and (almost) unbounded. For example, Windows uses scroll bars to overcome the limited display size on the width and the height. Components such as Tab controls and other means are used to benefit from the depth axis. RUIs make use of logical 3D spaces and, sometimes, of the display boundaries.

By convention, all 3d-numbers used in the 3-dimensional domain will apply the first value to the width axis, the second value to the height axis and the third value to the depth axis. For example, the 3d-number {5, 3, 90} refers to a 3D space of width 5, height 3 and depth 90.

Thus, three methods must exist to operate conversions between n-dimensional CUIs and 3D RUIs.

The method *3dnum getcapacity(dnum size, dirlist dirs)* gives the 3d-size of any d-size (type 3dnum refers to 3d-numbers and type dirlist refers to a list of directions – w for width, h for height, d for depth – such as {w, w, d, h}, corresponding to each dimension of size). For the sake of clarity, a d-size associated with a direction list will be referred to as a directional d-size. The method simply multiplies the values within size that have the same directions (for a direction never referred to within dirs, value 1 is used). For example, getcapacity({5, 4, 6, 2}, {w, d, w, w}) returns {60, 1, 4} and getcapacity({}, {}) returns {1, 1, 1}. Many dimensions may share the same direction. In the previous example, the width is composed of the first dimension which is 5 times the third which is 6 times the fourth of size 2, meaning 60 units (5x6x2 = 60).

The method *3dnum get3dcoord(dnum size, dnum index, dirlist dirs)* gives the corresponding 3d-coordinate of any d-coordinate within a d-size (for a direction never referred to within dirs, value 0 is used). For example, referring to FIG. 14A, get3dcoord({3, 7}, {2, 3}, {w, w}) returns {17, 0, 0} ({2x7 + 3, 0, 0}). Referring now to FIG. 14B, get3dcoord({5, 4, 6, 2}, {3, 1, 2, 0}, {w, d, w, w}) returns {40, 0, 1} ({3x6x2 + 2x2 + 0, 0, 1}). Obviously get3dcoord ({}, {}, {}) returns {0, 0, 0}.

The method *3dnum getndcoord(dnum size, 3dnum index, dirlist dirs)* is the opposite of the latter function. It gives the corresponding d-coordinate of any 3d-coordinate within a d-size (for a direction never referred to within dirs, value 0 is used). For example, referring to FIG. 14C, getndcoord({5, 4, 6, 2}, {31, 0, 3}, {w, d, w, w}) returns {2, 3, 3, 1}. Obviously getndcoord ({}, {0, 0, 0}, {}) returns {}.

Since the base size of a CC is directional (with the directions parameter), the images are by their nature bi-directional (width x height), the children of Container CCs are also directional (with the children direction parameter), text dimensions in Text CCs are still directional (by default conventions) and finally, the CUI stack is also directional (throughout the depth axis), by using the methods described above, any CUI can be converted to a RUI and fit in a 3D space.

One last method operates 3D accumulation, *3dnum 3dacc(3dnum sizes[], Direction dir).* It is used to accumulate (or pack) different 3d-sizes across one specified direction. The result will be a 3d-size that value across the direction will be the sum of all sizes values for the direction, and the two other values will be the maximal value of all sizes. For example, 3dacc({{4, 5, 1}, {3, 2, 8}, {0, 4, 3}}, height) will return {4, 11, 8}. The 4 is the maximal width of all 3 sizes, the 11 is the sum of all heights and the 8 is the maximal depth of all sizes.

Before being able to render CCs, some extra notions are needed, namely the matrix view, the element view and the user view, which are 3d-sizes.

The matrix view of a CC is the 3d-size of a rendered CC. It is usually different from one device to another, depending of the device characteristics (font size ...). It is calculated from the multiplication of the CC base d-size capacity (according to the CC directions)

and the element view, which is the 3d-size of a unit within the CC matrix (the calculation of element views is explained later). Mathematically, it is expressed as:

Matrix view = getcapacity(base d-size, directions) x Element view

It never takes into account the Container d-size of the CC, or where it is located in the CC tree. In other words, it ignores d-size accumulation. The Container CCs will take care of d-size accumulation in their own calculation of their element views. The interaction between the Matrix view and the Element view is presented schematically in FIG. 15A.

The element view of a CC is the 3d-size of one unit in the CC matrix. Its calculation, presented schematically in FIG. 15B, depends on the CC type (Control, Text or Container), except when images are defined in the CC (if their width and height are greater than 0). In this case:

Element view = {image width, image height, 1}

For Control CCs, the element view corresponds to the view of a single state. By convention, the element of a Control CC is only the current selection state, not the whole or a part of the States Matrix. Therefore, the element view is one state view. Since a state can be labeled or unlabeled, there are two ways to calculate the Control CC element view.

If State Bitmap Size is greater than 0 and the device accepts bitmaps:

Element view = {state bitmap width, state bitmap height, 1}

If State Bitmap Size is null and State String Max Length is greater than 0:

Element view = {state string max length x character width, character height, 1}

If both State Bitmap Size and State String Max Length are null, a selection is viewed as a state d-coordinate. For all values in the d-coordinate, the maximal number of digits must be calculated. For constant dimensions, the calculation is based on the States Matrix d-size value, but for variable dimensions, the number of digits is 10, based on the maximal long value (2147483648). Since the total digits is the sum of all values number of digits:

Element view = {total digits x character width, character height, 1}

There can be exceptions to this rule for some well-defined cases (for instance, constantly 2 unnamed states for a dimension, which is the definition of a switch box, which size can be calculated).

For Text CCs, considering that text view[0] is the first text view value (or 1 if not existing) and text view[1] is the second text view value (or 1 if not existing):

Element view = {text view[0] x character width, text view[1] x character height, 1}

For Container CCs, the element view is the 3D accumulation of the matrix views (that must have been previously calculated by using a recursive algorithm parsing the CC tree) of the child CCs across the children direction:

Element view = 3dacc(children matrix views, children direction}

The user view is a 3d-size, expressed in pixels, specifying the logical display, or what is immediately accessible by the end-user. For most devices, it is closely related to the

display size. For example, in a 1600x1200 Windows screen, it can simply be {1600, 1200, 1}. However, since Windows computers can hold a lot of memory and consequently, can save much more information than what is displayed on a single screen, and since Windows makes an extensive use of the depth direction, it would be preferable to set it at say {2400, 1800, 4} (9 times bigger than {1600, 1200, 1}). Although the physical screen size still holds {1600, 1200, 1} pixels, any part of the logical display can be shifted into it so the user can view all information. Referring to FIG. 15C, on a black-and-white 132x65 display Blackberry pager with limited RAM, a reasonable user view would be {200, 100, 1} (the depth direction is not scarcely used in the Blackberry compared to more user-friendly environments such as Windows).

5. Claims

What is claimed is:

1. A user interface, comprising:

a conceptual user interface_for defining interface requirements without reference to presentation elements, said definition being applicable to any platform; and

a rendered user interface for expressing said interface requirements on a pre-determined platform using the specific presentation elements associated therewith.

1.1 The interface according to claim 1, wherein said rendered user interface is configured for a windows-based platform.

1.2 The interface according to claim 1, wherein said rendered user interface is configured for a non-windows, hand-held device platform.

1.3 The interface according to claim 1.2, wherein said rendered user interface is configured for a paging device such as the Blackberry™.

2. The interface according to claim 1, wherein said conceptual user interface is defined using one or more conceptual components selected from the group consisting of:

(i) a text conceptual component (text CC), for defining one or more editable text fields;

(ii) a control conceptual component (control CC), for defining a domain of one or more states and enabling a user to make one or more selections over said domain; and

(iii) a container conceptual component (container CC), for logically grouping together one or more conceptual components, including container CCs.

3. The interface according to claim 1, wherein said conceptual user interface comprises:

a data structure, for ordering one or more conceptual components;

said conceptual components being selected from the group comprising:

(i) a text conceptual component (text CC), for defining one or more editable text fields;

(ii) a control conceptual component (control CC), for defining a domain of one or more states and enabling a user to make one or more selections over said domain; and

(iii) a container conceptual component (container CC), for logically grouping together one or more conceptual components, including container CCs.

4. The interface according to claim 3, wherein the data structure is a stack.

5. The interface according to claim 4, wherein conceptual components and null entries may be pushed onto and pulled from the stack.

6. The interface according to claim 5, wherein the position of a conceptual container in said stack is a factor used for said rendered user interface in expressing said interface requirements.

7. The interface according to claim 5, wherein user interaction is limited to the conceptual component at the top of said stack.

8. The interface according to claim 3, wherein said container CCs are employable to define a tree of conceptual components, and wherein the position of a conceptual component in said tree is a factor used for said rendered user interface in expressing said interface requirements.

9. The interface according to claim 8, wherein a tree defined from a plurality of conceptual components logically occupies a single position in said data structure.

10. The interface according to claim 3, wherein:

said data structure is a stack;

conceptual components and null entries may be pushed onto and pulled from the stack; and

user interaction is limited to the conceptual component at the top of the stack.

11. The interface according to claim 3, wherein each type of conceptual component is associated with a multi-dimensional matrix of values, wherein said dimensions include the zero dimension.

12. The interface according to claim 11, wherein the units of said values matrix are homogenate.

13. The interface according to claim 11, wherein each unit of said values matrix for said text CC holds text.

14. The interface according to claim 13, wherein said text CC is associated with a data member indicating a number of dimensions for all elements of the corresponding values matrix, zero dimensions being rendered as a single character; a single dimension being rendered as a line of text; two dimensions being rendered as a plurality of lines of text.

15. The interface according to claim 11, wherein said values matrix for said control CC holds user-entered selections from said domain of states.

15.1 The interface according to claim 15, wherein:

said domain of states is defined by a multi-dimensional matrix, including the zero dimension, which represents states; and

said user-entered selections are denoted as co-ordinates of said state matrix.

15.2 The interface according to claim 15.1, wherein:

said states matrix is associated with a plurality of labels which are used to render the states to the end-user.

15.3 The interface according to claim 15.1, wherein the states matrix is associated with an indicator representing a maximum string length for any said rendering of said labels.

15.4 The interface according to claim 15.3, wherein:

said states matrix is associated with a plurality of graphics which are used to render the states to the end-user.

15.5 The interface according to claim 15.4, wherein, in the event said graphic size is zero and said maximum string length indicator is zero, the states are rendered anonymously.

16. The interface according to claim 11, wherein each unit of said values matrix for said container CC holds a list of zero or more of said conceptual components.

16.1 The interface according to claim 16, wherein said container CC is associated with a set of zero or more child conceptual components, instances of which are used to populate each unit of said values matrix.

16.2 The interface according to claim 16.1, wherein said container CC includes Cartesian directional informational that is optionally used to visually render the child conceptual components.

17. The interface according to claim 11, wherein each type of conceptual component is associated with a multi-dimension base size, said dimensions including a zero dimension.

18. The interface according to claim 17, wherein said container CCs include means for defining a tree of conceptual components.

19. The interface according to claim 18, wherein the dimension size of said values matrix for a given conceptual component that is a child of a parent conceptual component in said tree is an accumulation of the values dimension size of its parent conceptual component and the base dimension size of said given conceptual component.

20. The interface according to claim 11, wherein each conceptual component type can optionally be associated with a matrix of images, said images being used in expressing said interface requirements unless said platform is incapable of rendering said images.

21. The interface according to claim 19, wherein:

each conceptual component type can optionally be associated with a matrix of images, said images being used in expressing said interface requirements unless said platform is incapable rendering said images; and

the dimension size of said images matrix for said given conceptual component is equal to the dimension size of said values matrix.

23. The interface according to claim 11, wherein each conceptual component type includes a matrix of read-only attributes for specifying whether or not elements of said values matrix enable user input.

24. The interface according to claim 19, wherein:

each conceptual component type includes a matrix of read-only attributes for specifying whether or not elements of said values matrix enable user input; and

the dimension size of said read-only attributes matrix for said given conceptual component is equal to the dimension size of said values matrix.

24.1 The interface according to claim 23 or 24, wherein said matrix of read-only attributes is associated with a flag which indicates that the entire conceptual component is read-only.

25. The interface according to claim 11, wherein each conceptual component type includes a labels matrix for describing said values matrix to the end user.

26. The interface according to claim 19, wherein:

each conceptual component type includes a labels matrix for describing said values matrix to the end user; and

the dimension of said labels matrix for said given conceptual component is not accumulative.

26.1 The interface according to claim 26, wherein the dimension of said labels matrix is equal to said base dimension size.

26.2 The interface according to claim 15.1, wherein the labels matrix is associated with an indicator representing a maximum string length for any rendering of said labels.

26.3 The interface according to claim 26.2, wherein:

said states matrix is associated with a plurality of graphics which are used to render the values of said values matrix to the end-user.

26.4 The interface according to claim 26.3, wherein, in the event said graphic size is zero and said maximum string length indicator is zero, the values of said values matrix are rendered anonymously.

27. The interface according to any of claims 17-19, including means for fixing the base dimension size or enabling it to vary.

28. The interface according to claim 27, including means for varying the base-dimension size.

29. The interface according to claim 28, including means for enabling or preventing the end-user from varying the base-dimension size.

29.1 The interface according to claim 29, means for associating a Cartesian direction for each dimension of said values matrix.

30. The interface according to claim 18, including means for enabling application logic to render one selected child conceptual at a time that the end-user can interact with, or preventing same.

Application number / numéro de demande: 2388101

Figures: 7B — 13-15A — 15B - 15-C-21
11 - 12B -

Unscannable items
received with this application
(Request original documents in File Prep. Section on the 10th floor)

Documents reçu avec cette demande ne pouvant être balayés
(Commander les documents originaux dans la section de préparation des dossiers au 10ème étage)

Conceptual User Interface

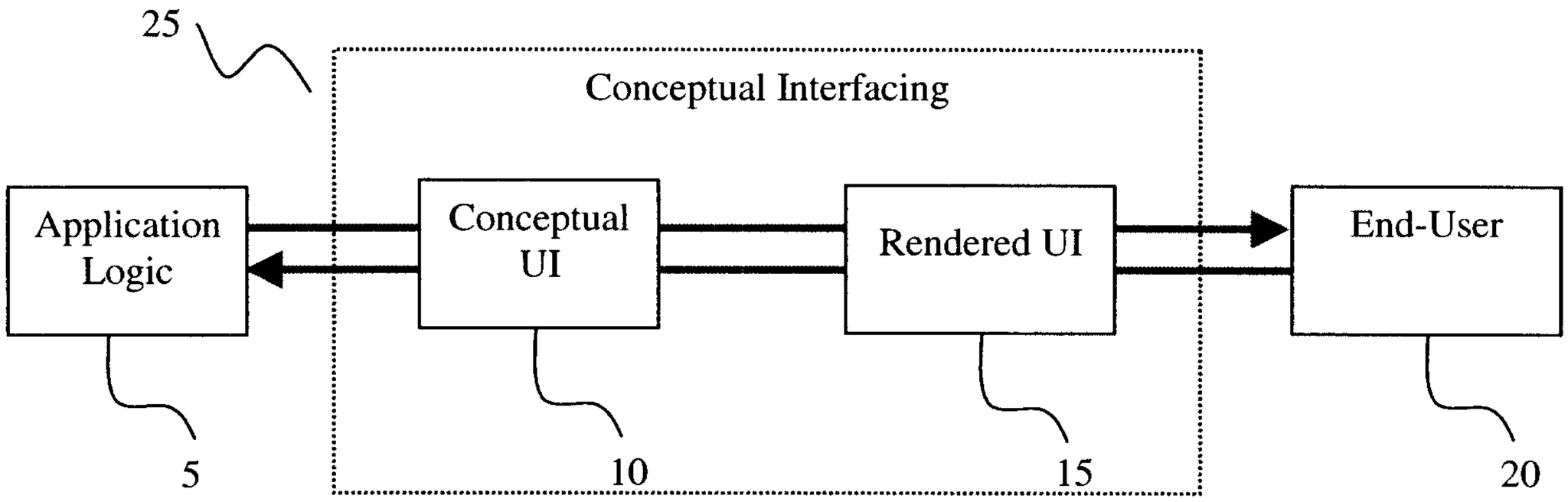


FIG. 1

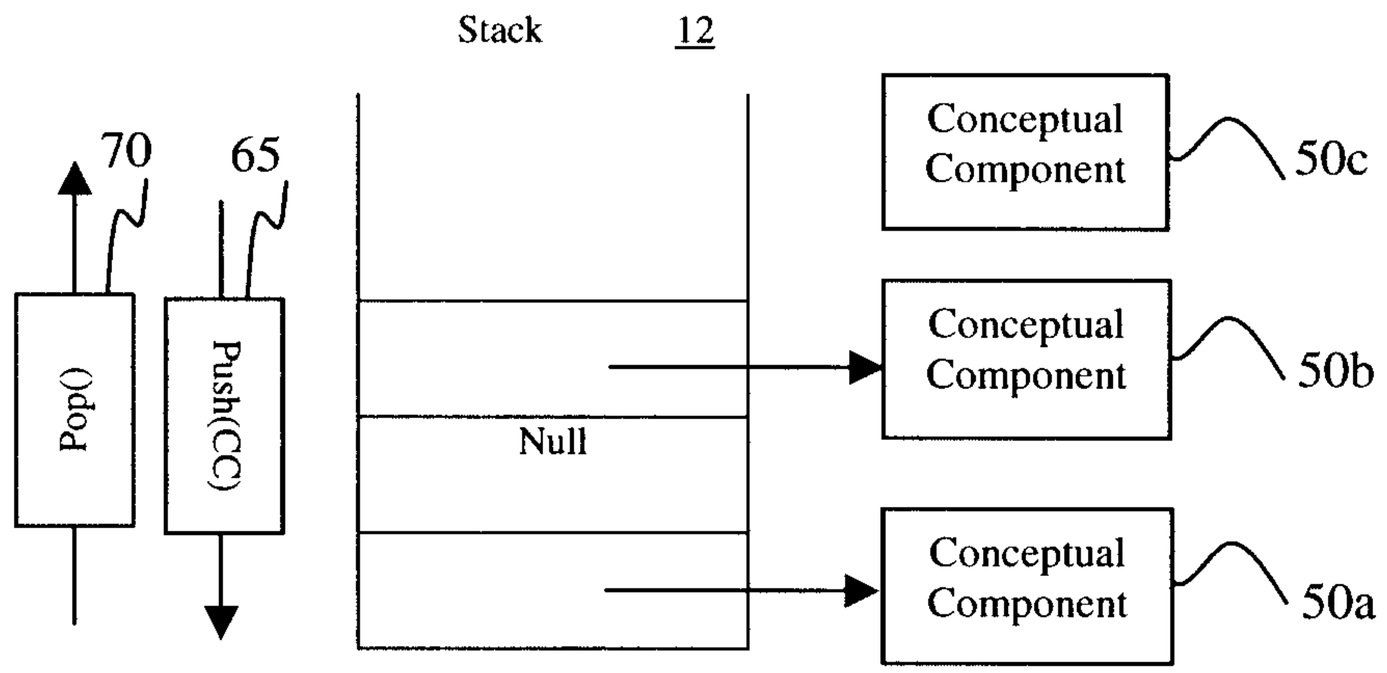


FIG. 2

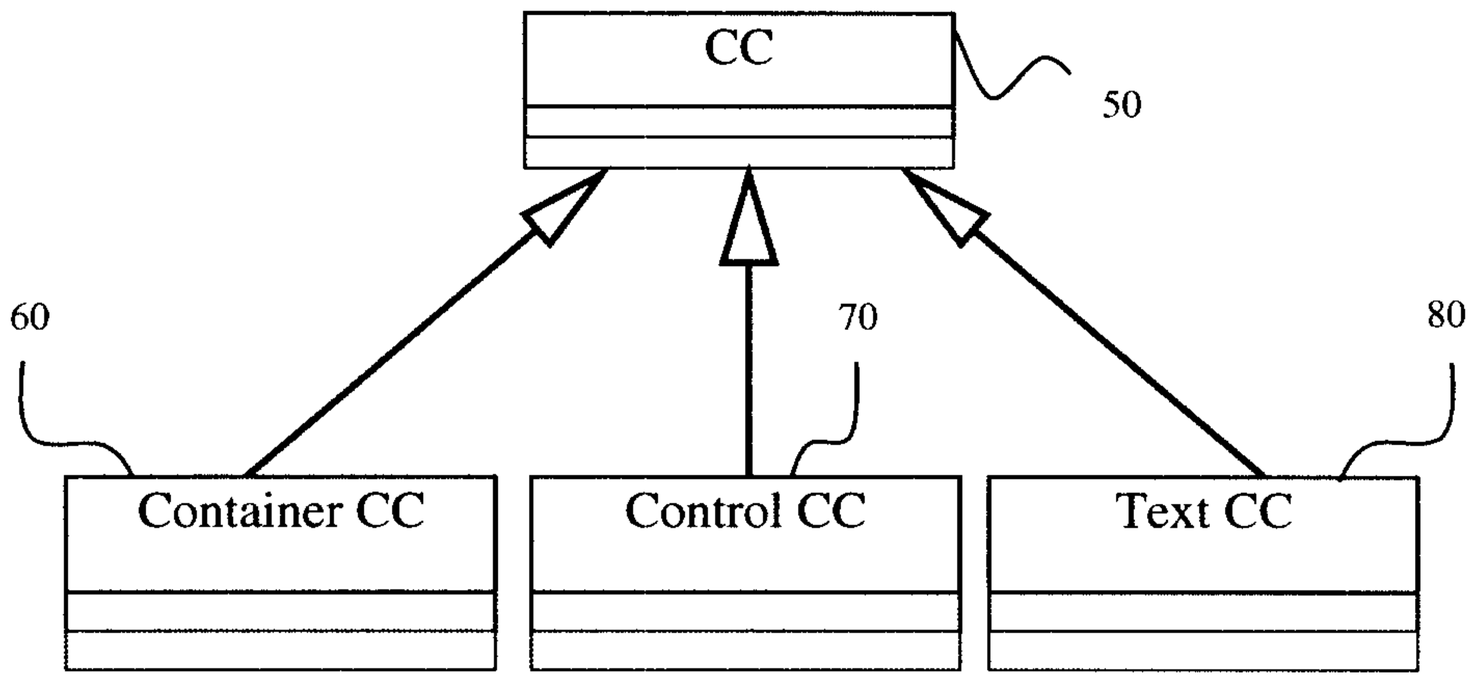


FIG. 3

CC			50
Data	105	Modification Methods	110

Conceptual User Interface

CC			50
Data	105	Modification Methods	110
Base d-size	150	Initialization time int insert(dnum start, dnum range) 155 int remove(dnum start, dnum range) 160	
Constant Flags	175		
End-User Resize Flags	180		
Directions	185		
Default Action Flag	190		
Semantic Type	195	Initialization time	
User-Defined Flags	200		
Labels Matrix 115 — String Max Length	215		
Labels Matrix 115 — Bitmap Size	220		
Labels Matrix 115 — Buffered Storage System OR	225		
Labels Matrix 115 — Default Storage System OR	230	int setlabel(char *label, dnum index = dnumnull) 235 int setbitmap(Bitmap *bitmap, dnum index = dnumnull) 240	
Labels Matrix 115 — User-Defined Storage System	245	int updatelabels(dnum pos, dnum range) 250	
Read-Only Flags Matrix 120 — Constant Flag	270	Initialization time	
Read-Only Flags Matrix 120 — Default Storage System OR	275	int setroflag(boolean flag, dnum index = dnumnull) 280	
Read-Only Flags Matrix 120 — User-Defined Storage System	285	int updateroflags(dnum pos, dnum range) 290	
Images Matrix 125 — 2D size (width and height)	320	Initialization time	
Images Matrix 125 — Default Storage System	325	Text & Graphical Commands (line, circle, bitmap generation, setfont, setpos… and dnum index = dnumnull as last parameter) 330	
Images Matrix 125 — User-Defined Storage System	335	int updatevalues(dnum pos, dnum range) 340	

FIG. 4

Conceptual User Interface

Container CC			60
Data	105	Modification Methods	110
Children Set	365	Initialization time	
Children Direction	370		
Select CC Flag	375		
Selected CC	378	int selectcc(int childindex)	385
Instances Matrix 360	Matrix of instances of Children Set **365**	int updatevalues(dnum pos, dnum range)	395

FIG. 5A

Control CC			70
Data	105	Modification Methods	110
States Matrix	405	Initialization time Matrix-related methods (insert, remove, setlabel, setbitmap, …)	410
Selections Matrix 400	Default Storage System 420	int setselection(dnum sel, dnum index = dnumnull)	425
	User-Defined Storage System 430	int updatevalues(dnum pos, dnum range)	435

FIG. 5B

Text CC			80
Data	105	Modification Methods	110
Number of Dimensions	455	Initialization time	
Maximal Length	460		
Text View	465		
Texts Matrix 450	Default Storage System 470	int settext(dnum pos, long lengthtoremove, char *string, long stringlength, dnum index = dnumnull)	475
	User-Defined Storage System 480	int updatevalues(dnum pos, dnum range)	485

FIG. 5C

Conceptual User Interface

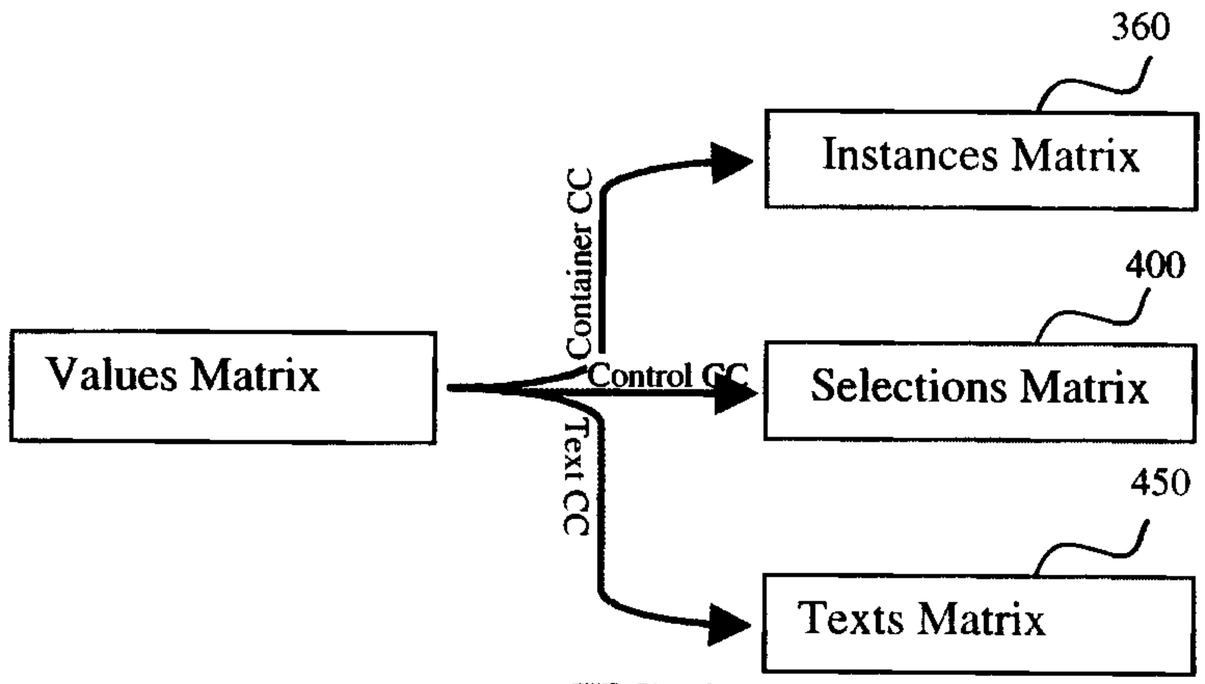


FIG. 6

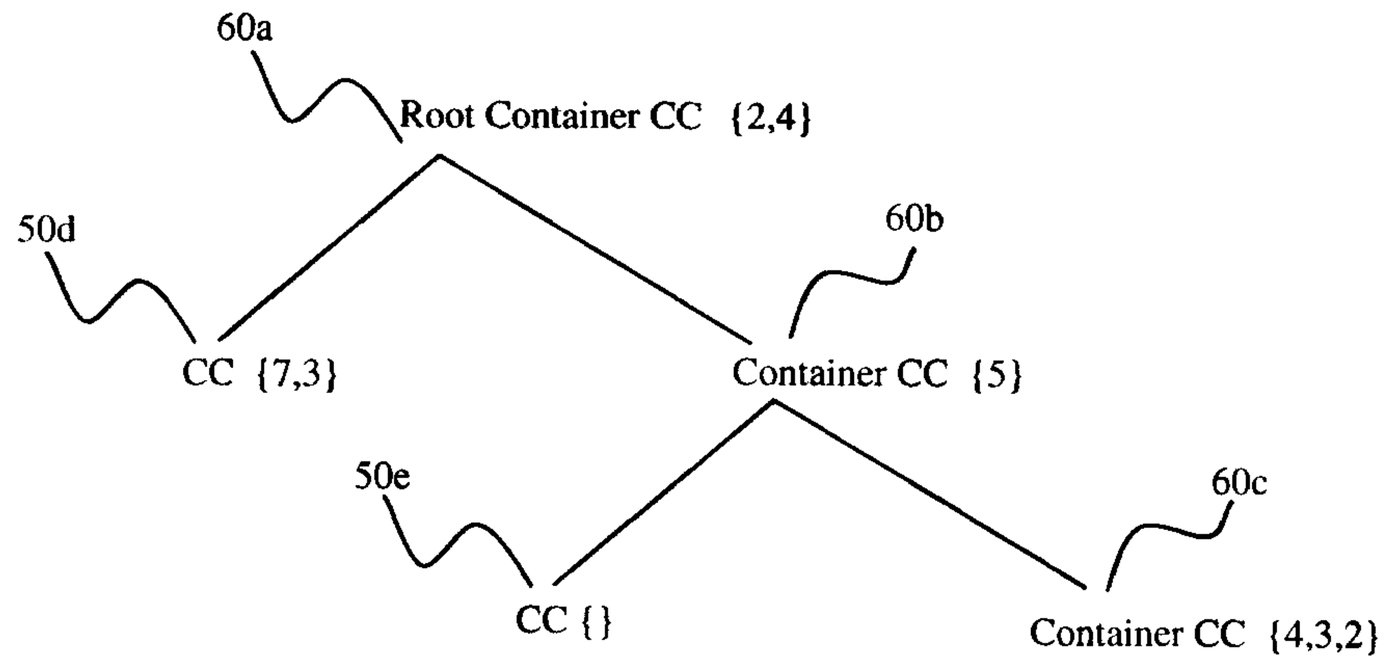


FIG. 7A

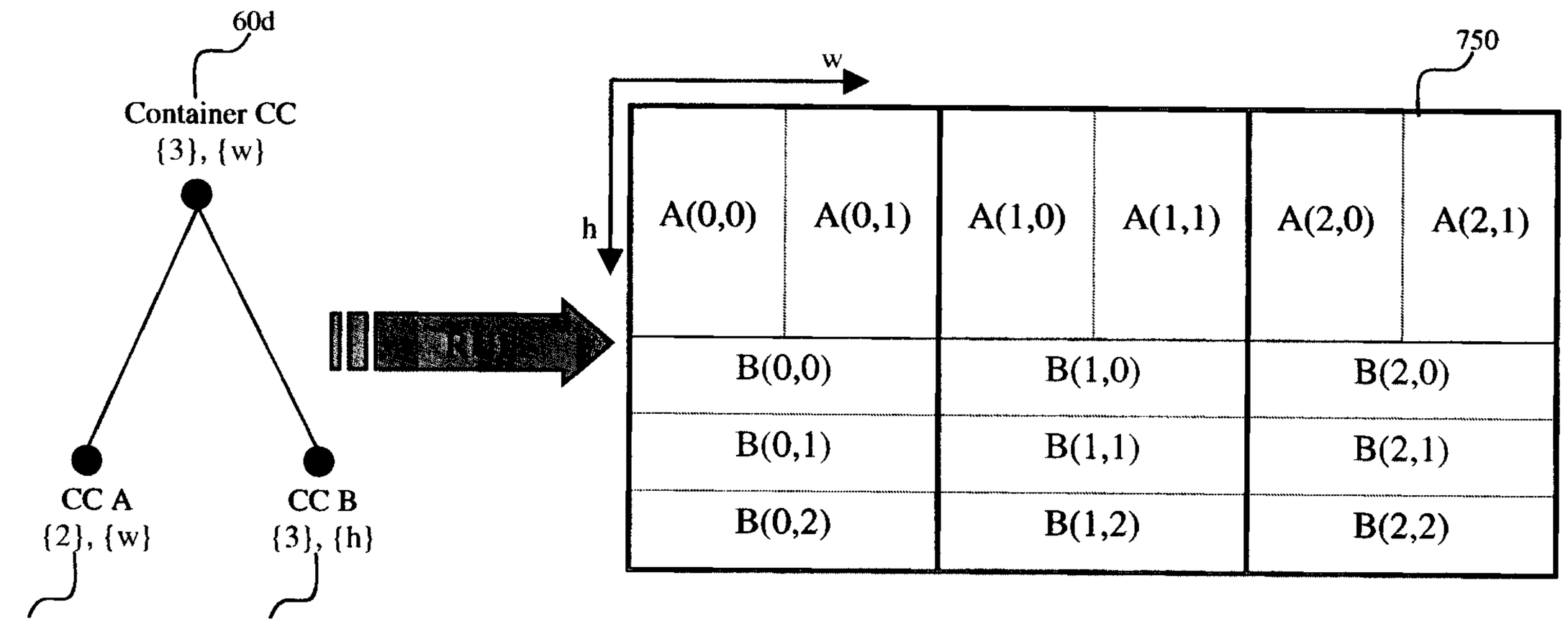


FIG. 7B

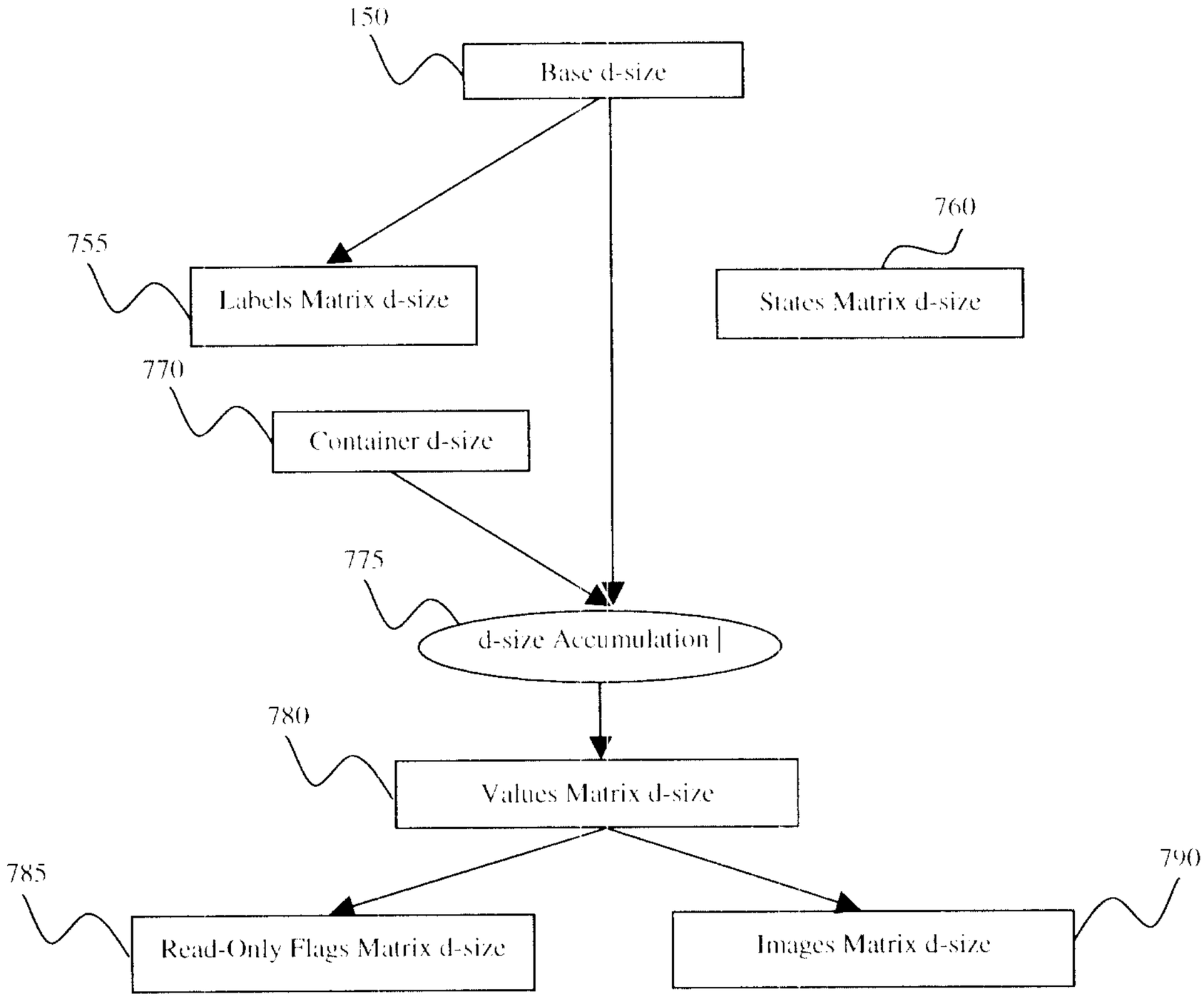

150
Base d-size
755
Labels Matrix d-size
760
States Matrix d-size
770
Container d-size
775
d-size Accumulation
780
Values Matrix d-size
785
Read-Only Flags Matrix d-size
790
Images Matrix d-size

FIG. 8

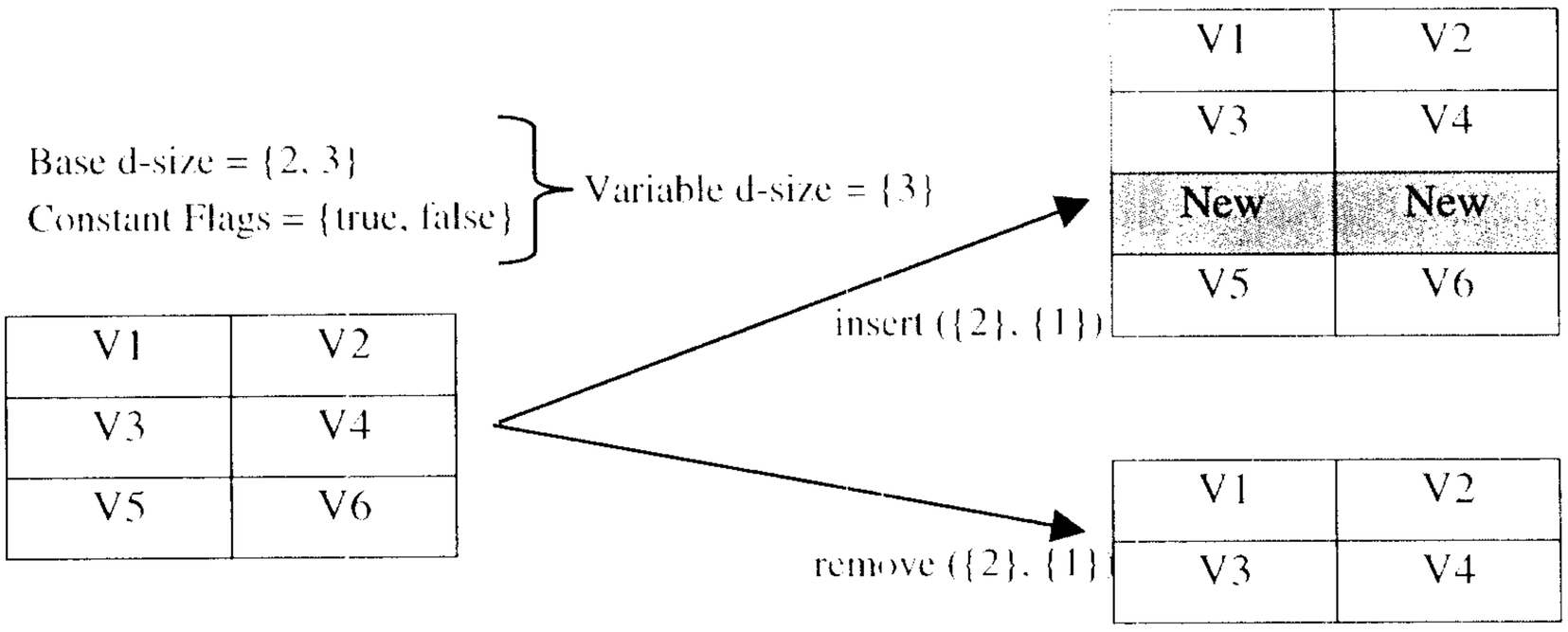

Base d-size = {2, 3}
Constant Flags = {true, false}
Variable d-size = {3}
V1
V2
V3
V4
V5
V6
insert ({2}, {1})
New
New
remove ({2}, {1})

FIG. 9

Control CC 70		
Data Members 105		Value 505
Base d-size 150		{2}
Constant Flags 175		{TRUE}
End-User Resize Flags 180		{FALSE}
Directions 185		{w}
Default Action Flag 190		TRUE
Semantic Type 195		""
User-Defined Flags 200		All 3 FALSE
Labels Matrix 115	String Max Length 215	3
	Bitmap Size (width and height) 220	{0, 0}
	Strings (units of Labels Matrix **115**, according to Buffered Storage System **225**)	"me" "you"
Read-Only Matrix 120	Constant Flag 270	FALSE
Images Matrix 125	2D size (width and height) 320	{0, 0}
States Matrix 405	State d-size	{4}
	Constant Flags	{TRUE}
	End-User Resize Flags	{FALSE}
	Directions	{d}
	String Max Length	1
	Bitmap Size (width and height)	{0, 0}
	Strings (units of States Matrix 405)	"A" "B" "C" "D"
Selections Matrix 400	End-User Selections (Default Storage System **420**)	{1} {3}

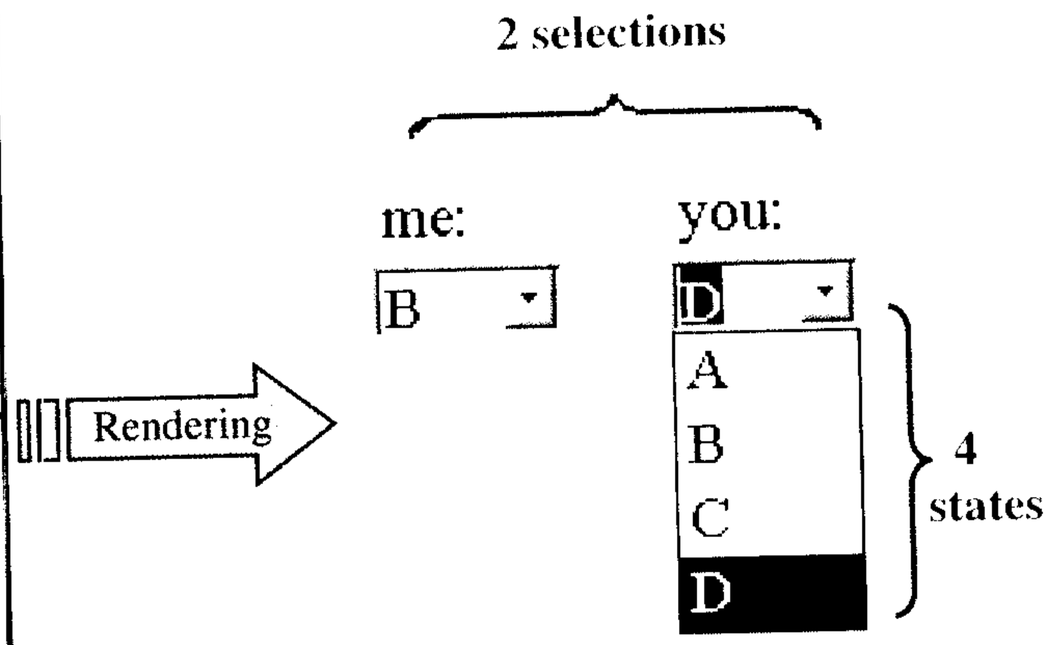


FIG. 10

States Matrix d-size 760

	Bitmap Size 220			
	= 0		> 0	
	String Max Length 215			
	= 0	> 0	= 0	> 0
≠		A		Run Notepad
{4}		B		yellow, red, green, magenta
{2,3}		A B C D E F		black, gray, blue, orange, red, green

FIG. 11

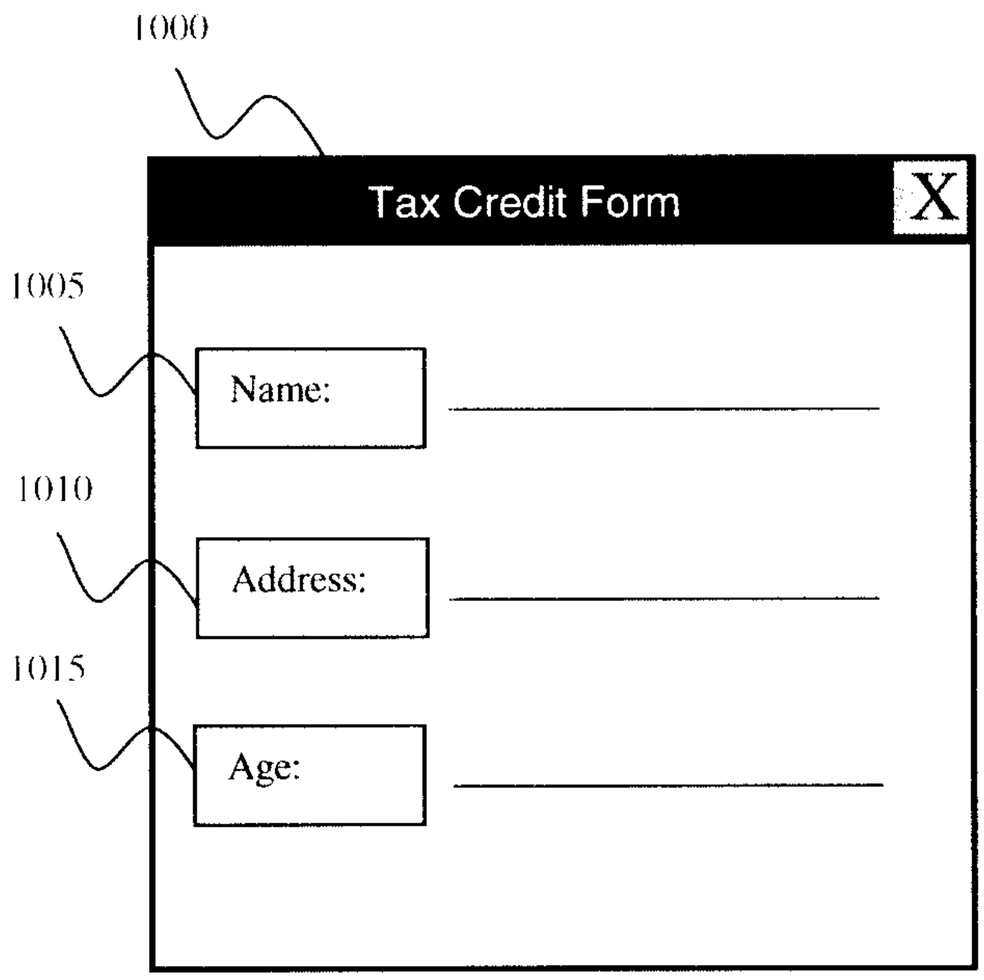


FIG. 12A

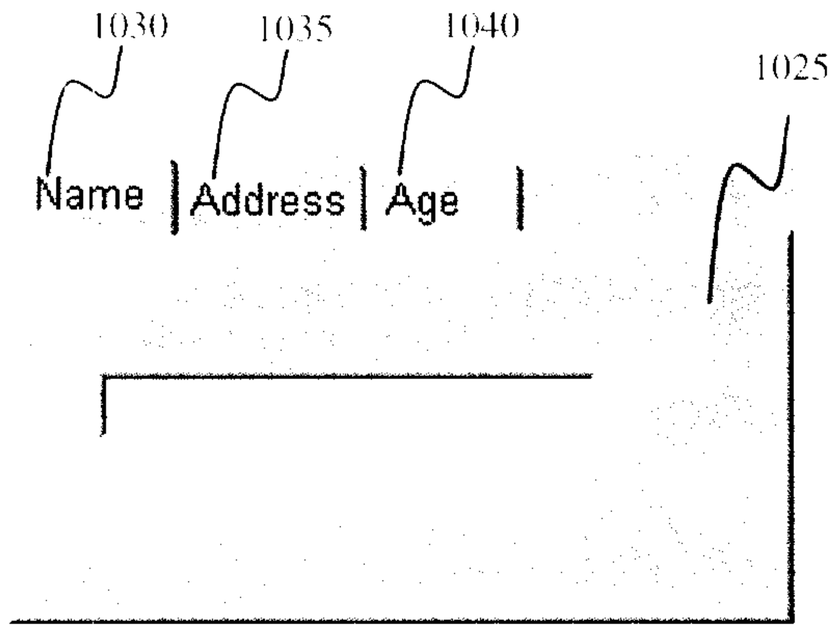
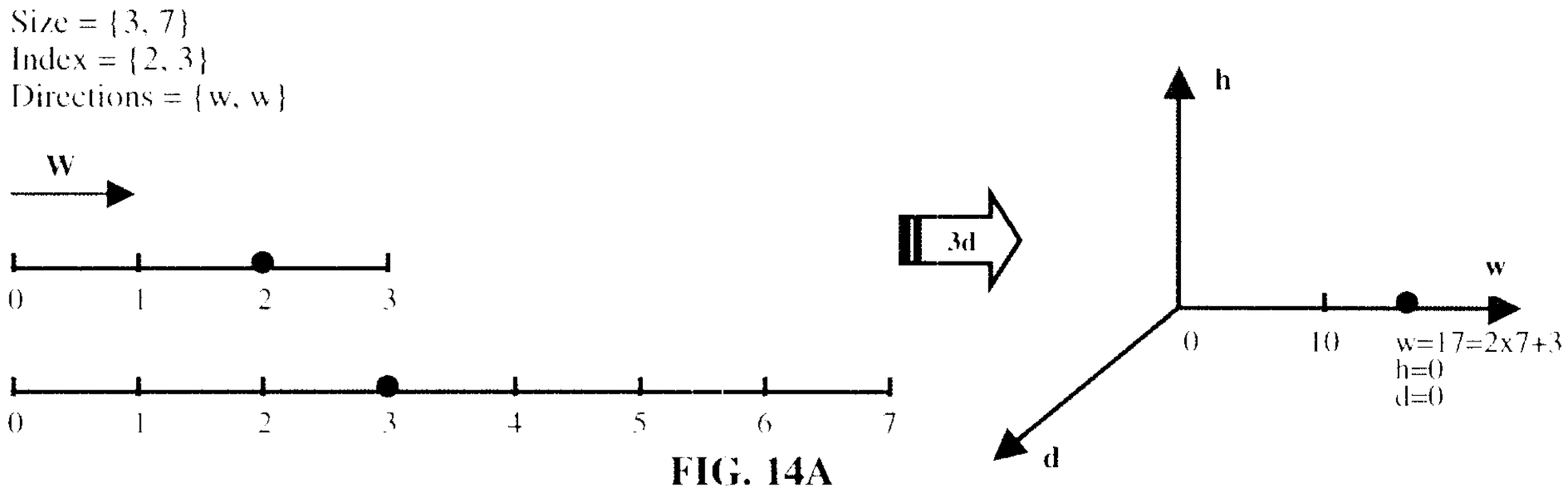


FIG. 12B



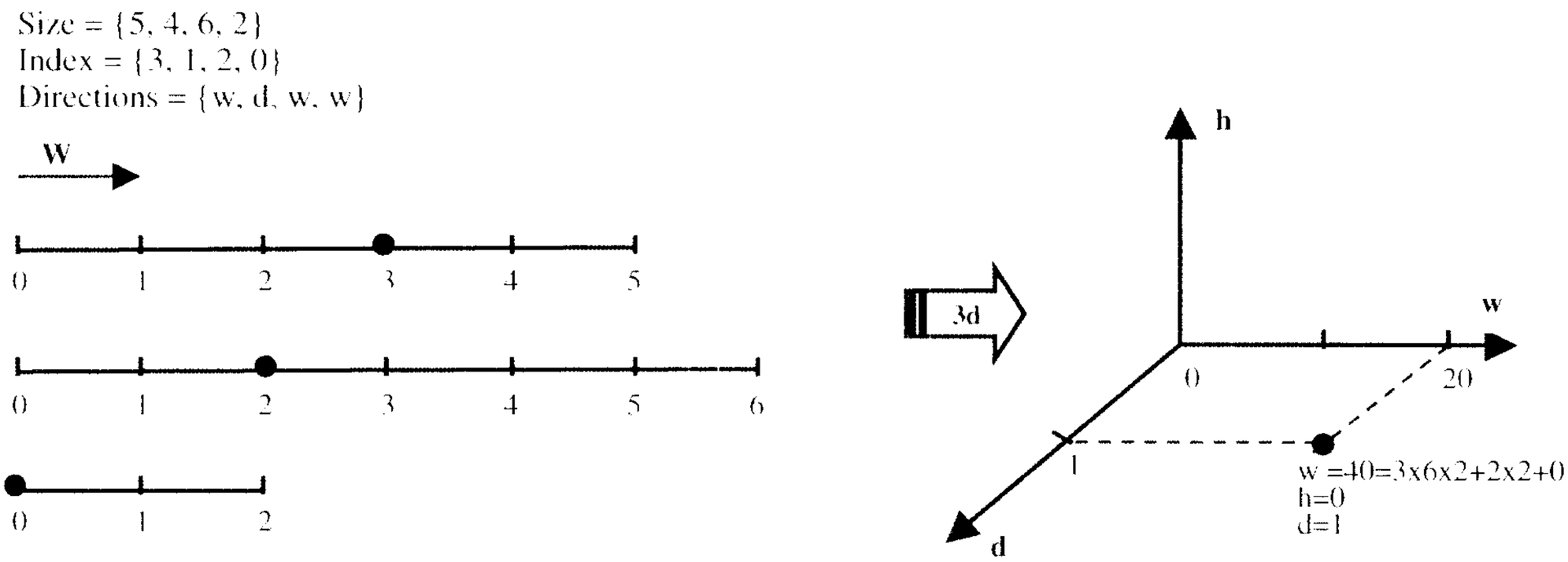


FIG. 14B

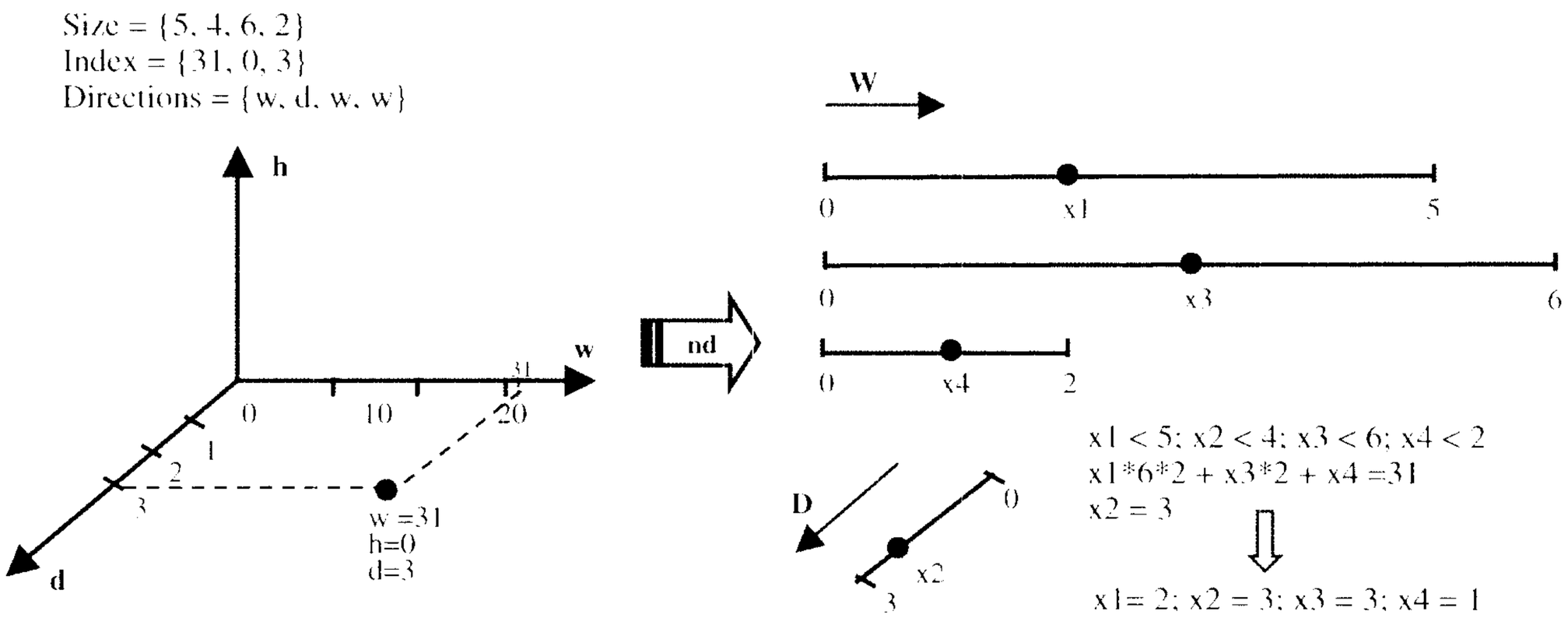


FIG. 14C

Conceptual User Interface

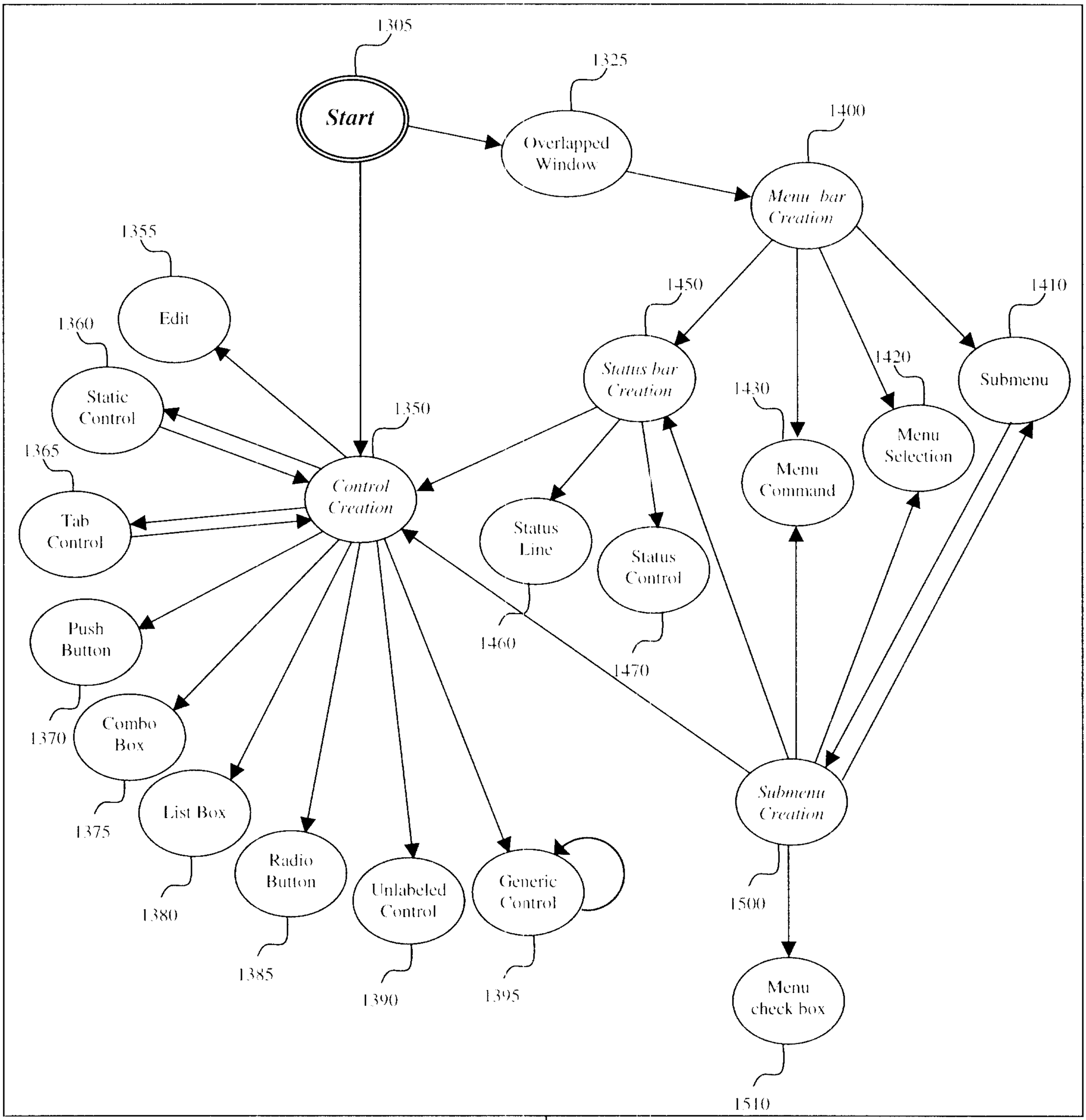


FIG. 17

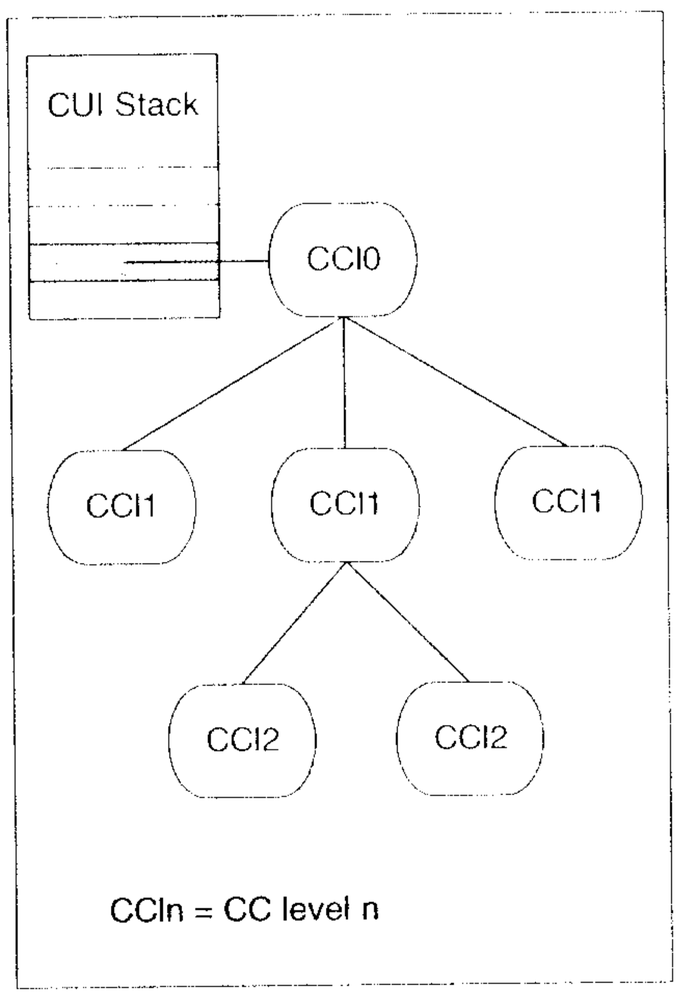


FIG. 18A

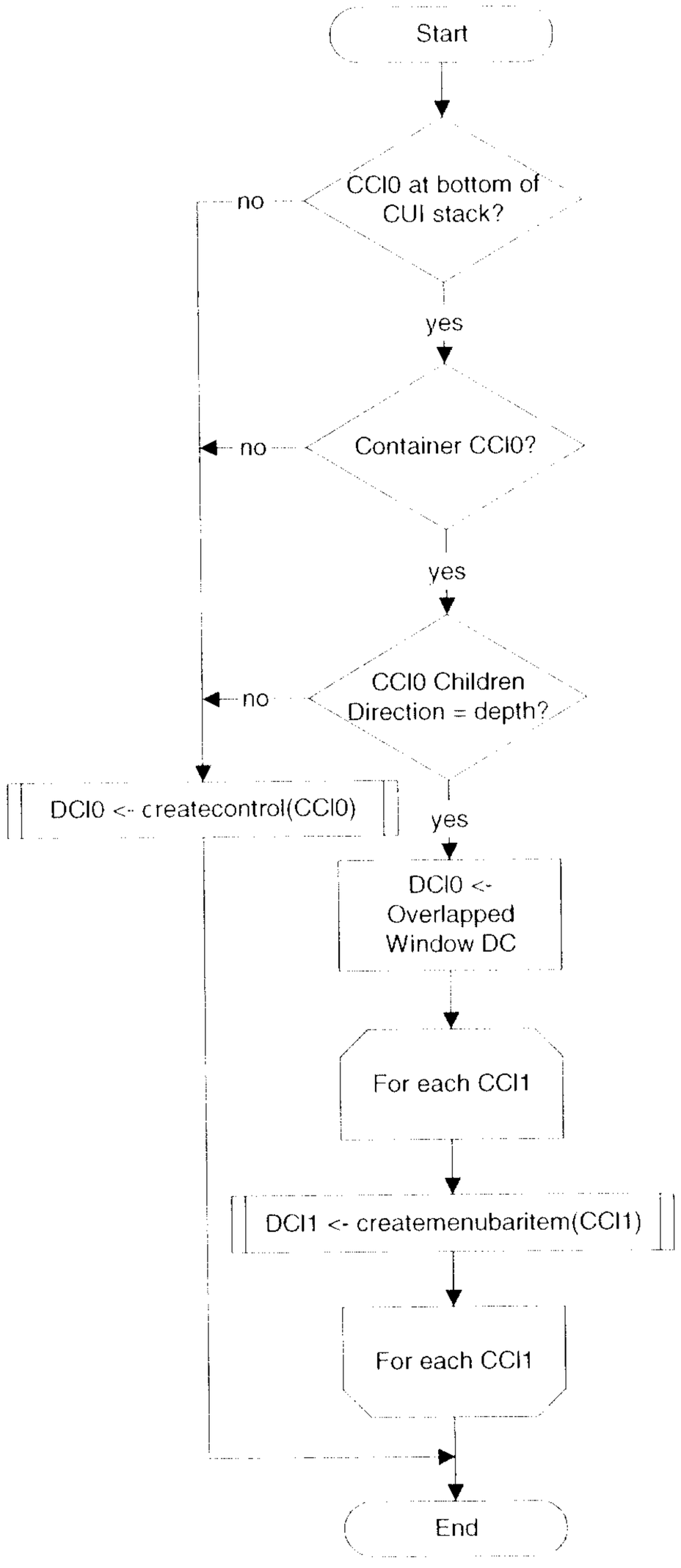


FIG. 18B

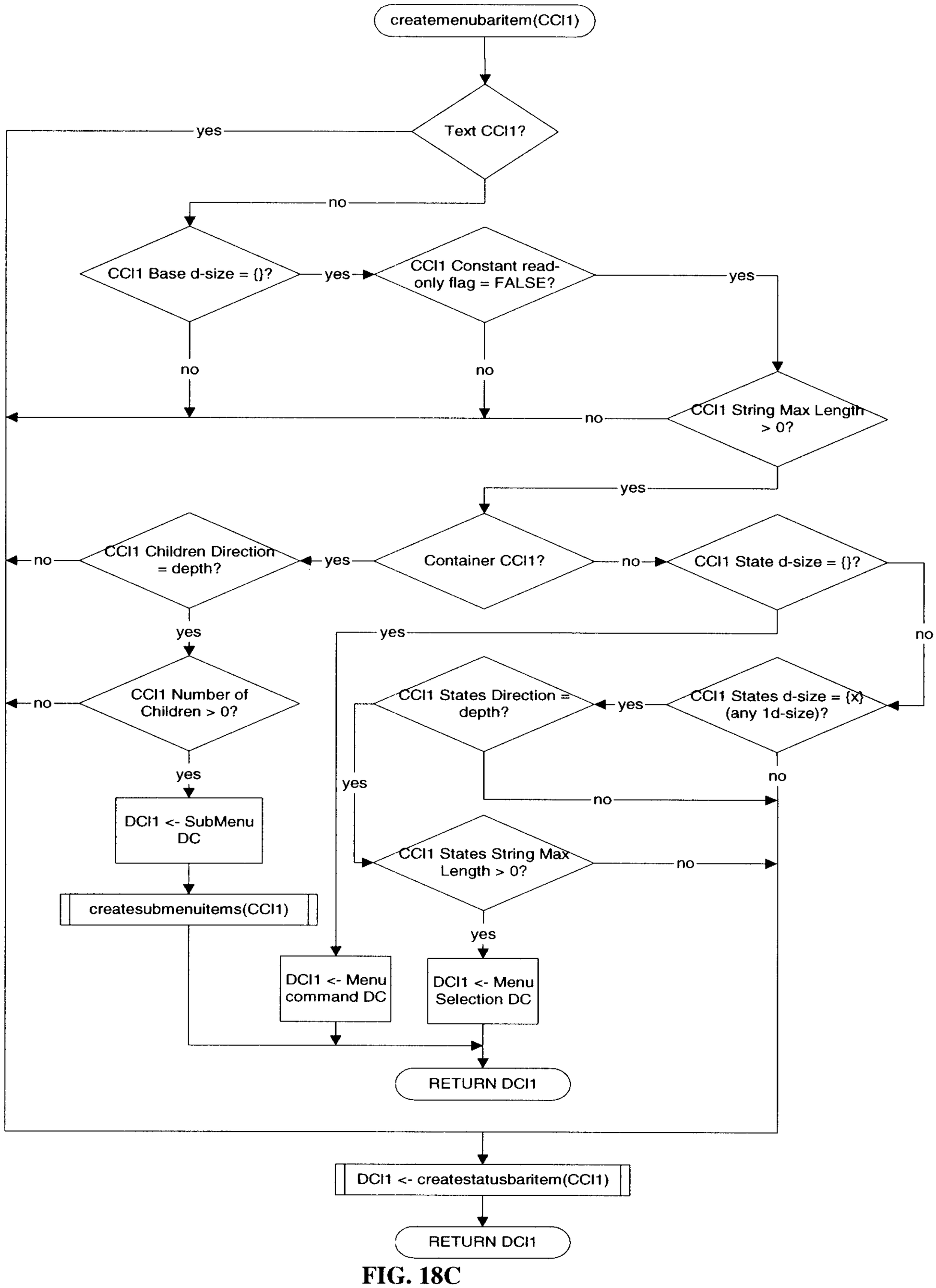

createmenubaritem(CCI1)
Text CCI1?
yes
no
CCI1 Base d-size = {}?
yes
CCI1 Constant read-only flag = FALSE?
yes
no
no
no
CCI1 String Max Length > 0?
yes
no
CCI1 Children Direction = depth?
yes
Container CCI1?
no
CCI1 State d-size = {}?
no
yes
yes
CCI1 Number of Children > 0?
CCI1 States Direction = depth?
yes
CCI1 States d-size = {x} (any 1d-size)?
no
yes
yes
no
no
DCI1 <- SubMenu DC
CCI1 States String Max Length > 0?
no
createsubmenuitems(CCI1)
yes
DCI1 <- Menu command DC
DCI1 <- Menu Selection DC
RETURN DCI1
DCI1 <- createstatusbaritem(CCI1)
RETURN DCI1

FIG. 18C

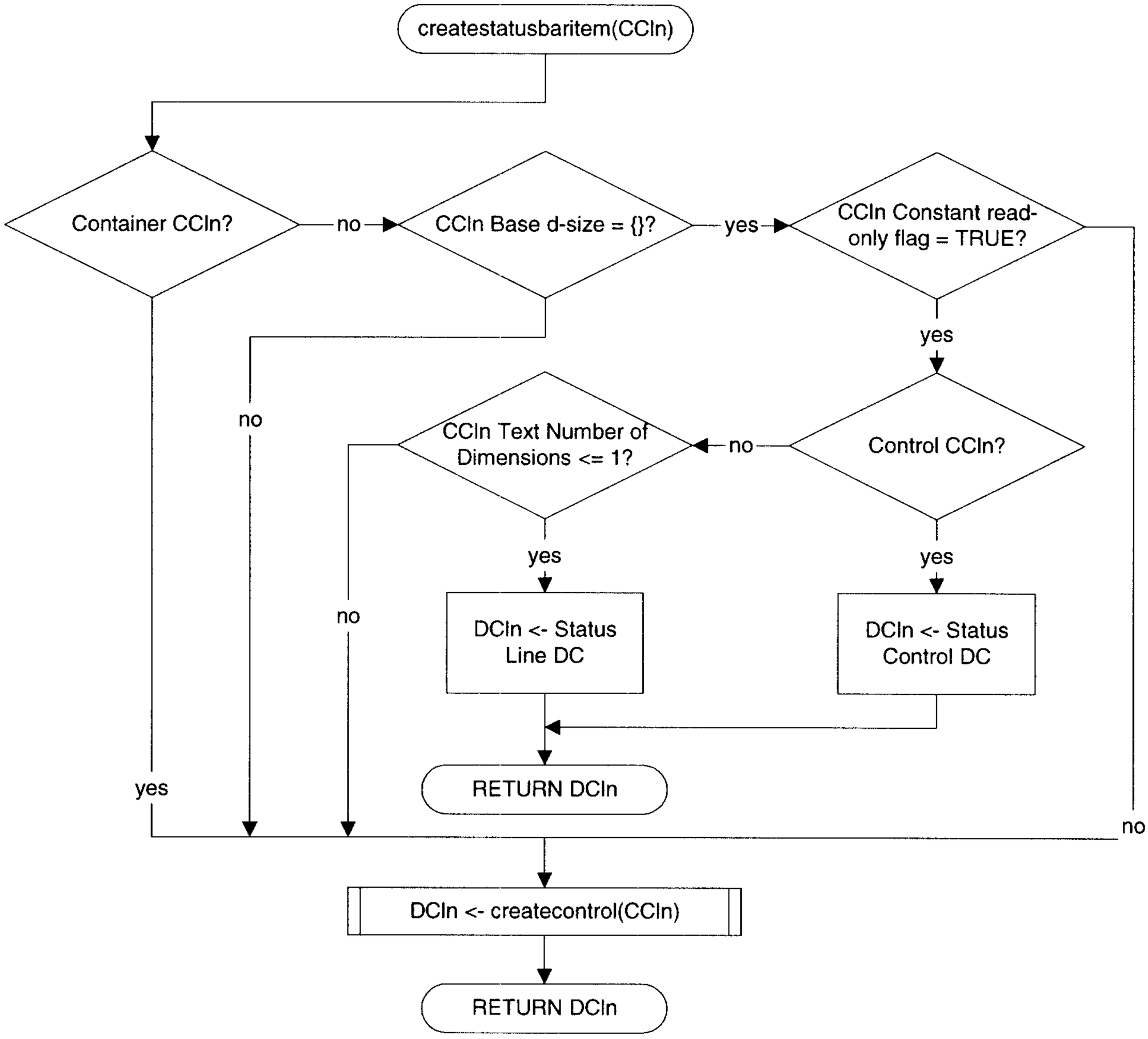

createstatusbaritem(CCIn)
Container CCIn?
no
CCIn Base d-size = {}?
yes
CCIn Constant read-only flag = TRUE?
yes
yes
CCIn Text Number of Dimensions <= 1?
no
Control CCIn?
yes
yes
no
no
DCIn <- Status Line DC
DCIn <- Status Control DC
RETURN DCIn
yes
no
DCIn <- createcontrol(CCIn)
RETURN DCIn

FIG. 18D

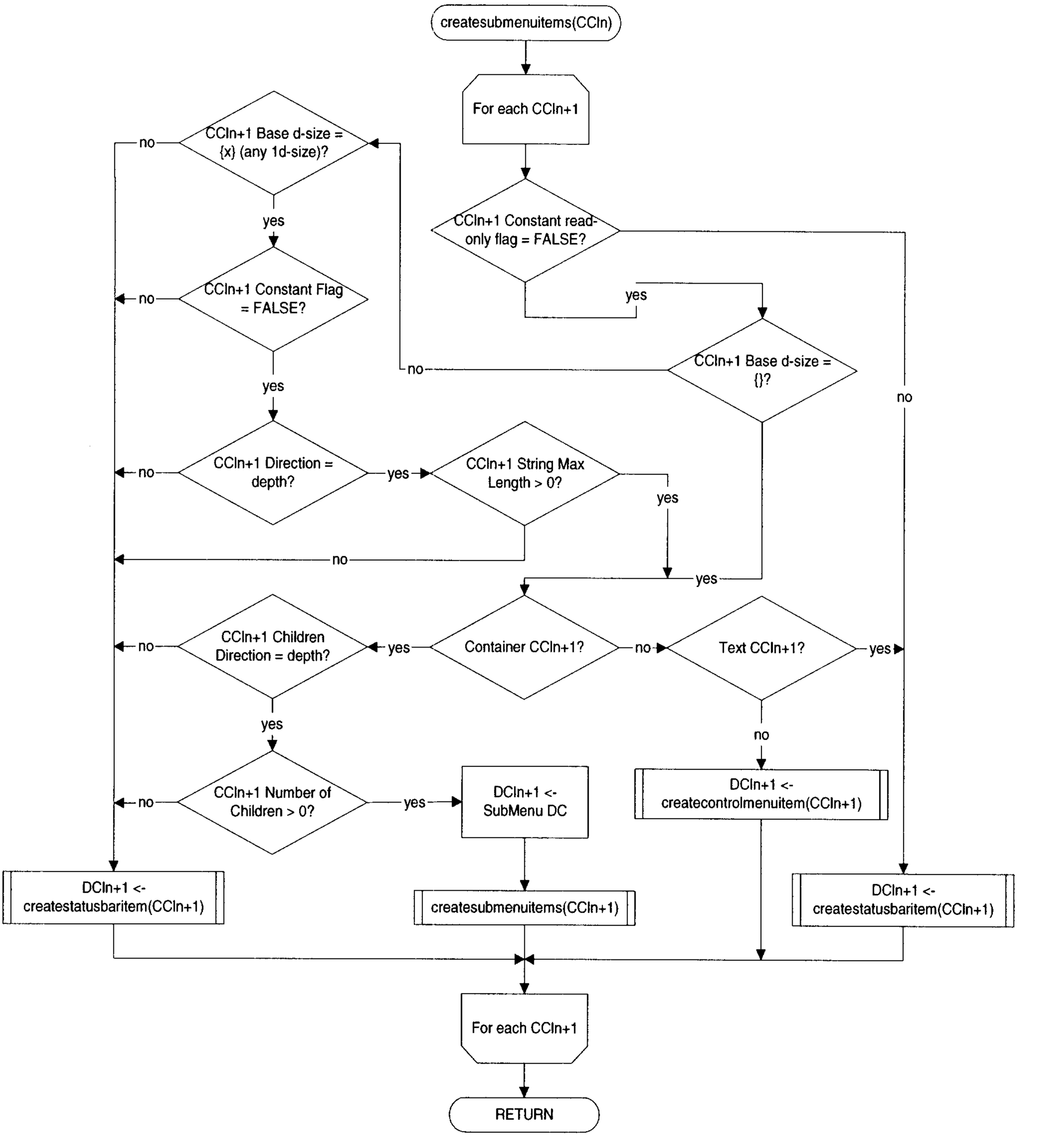

createsubmenuitems(CCIn)
For each CCIn+1
CCIn+1 Constant read-only flag = FALSE?
yes
no
CCIn+1 Base d-size = {}?
no
yes
CCIn+1 Base d-size = {x} (any 1d-size)?
no
yes
CCIn+1 Constant Flag = FALSE?
no
yes
CCIn+1 Direction = depth?
no
yes
CCIn+1 String Max Length > 0?
yes
no
yes
CCIn+1 Children Direction = depth?
no
yes
Container CCIn+1?
yes
no
Text CCIn+1?
yes
no
yes
CCIn+1 Number of Children > 0?
no
yes
DCIn+1 <- SubMenu DC
DCIn+1 <- createcontrolmenuitem(CCIn+1)
DCIn+1 <- createstatusbaritem(CCIn+1)
DCIn+1 <- createstatusbaritem(CCIn+1)
createsubmenuitems(CCIn+1)
For each CCIn+1
RETURN

FIG. 18E

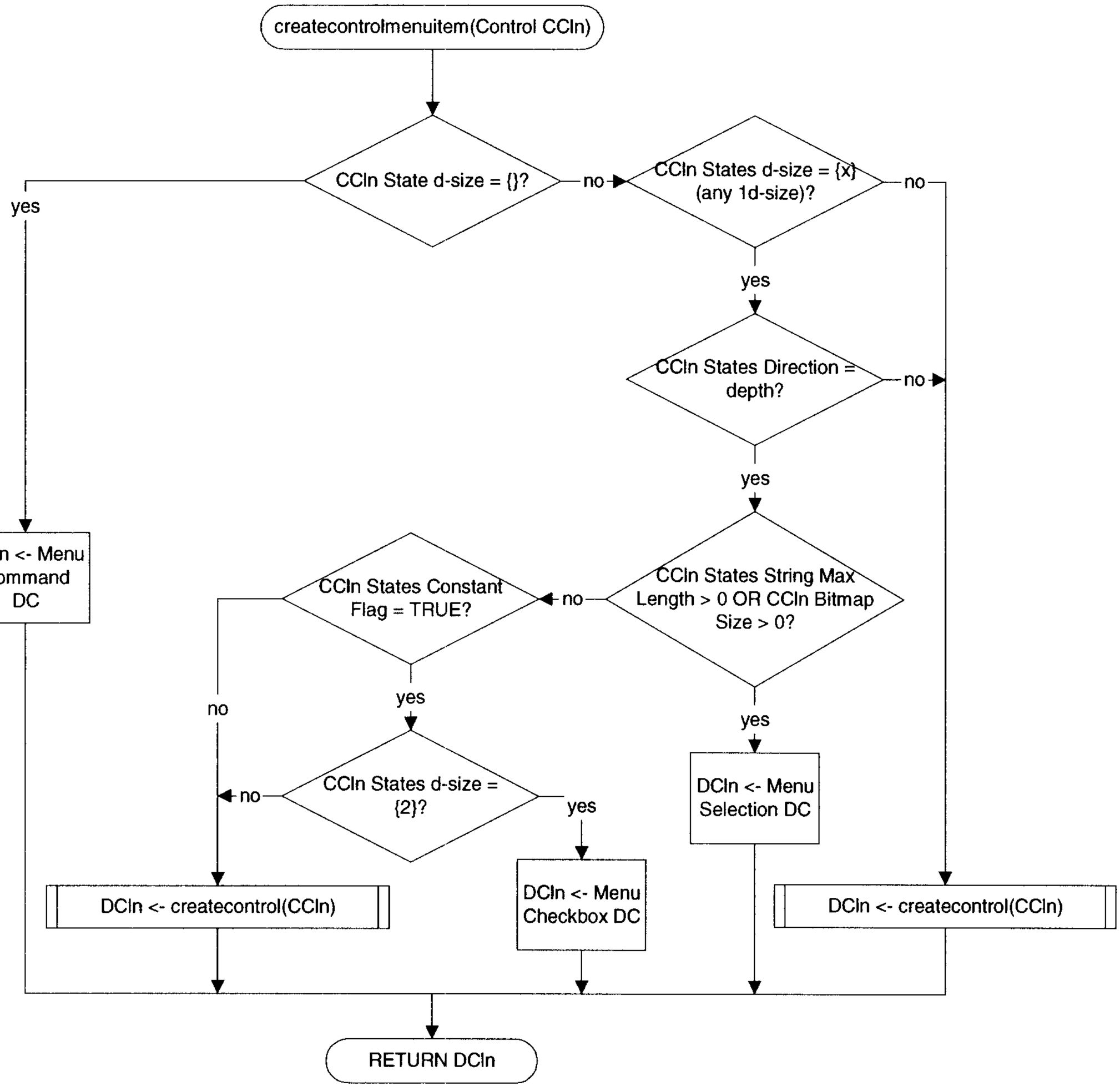

createcontrolmenuitem(Control CCIn)
CCIn State d-size = {}?
no
yes
CCIn States d-size = {x} (any 1d-size)?
no
yes
CCIn States Direction = depth?
no
yes
CCIn States String Max Length > 0 OR CCIn Bitmap Size > 0?
no
yes
CCIn States Constant Flag = TRUE?
no
yes
CCIn States d-size = {2}?
no
yes
DCIn <- Menu Command DC
DCIn <- createcontrol(CCIn)
DCIn <- Menu Checkbox DC
DCIn <- Menu Selection DC
DCIn <- createcontrol(CCIn)
RETURN DCIn

FIG. 18F

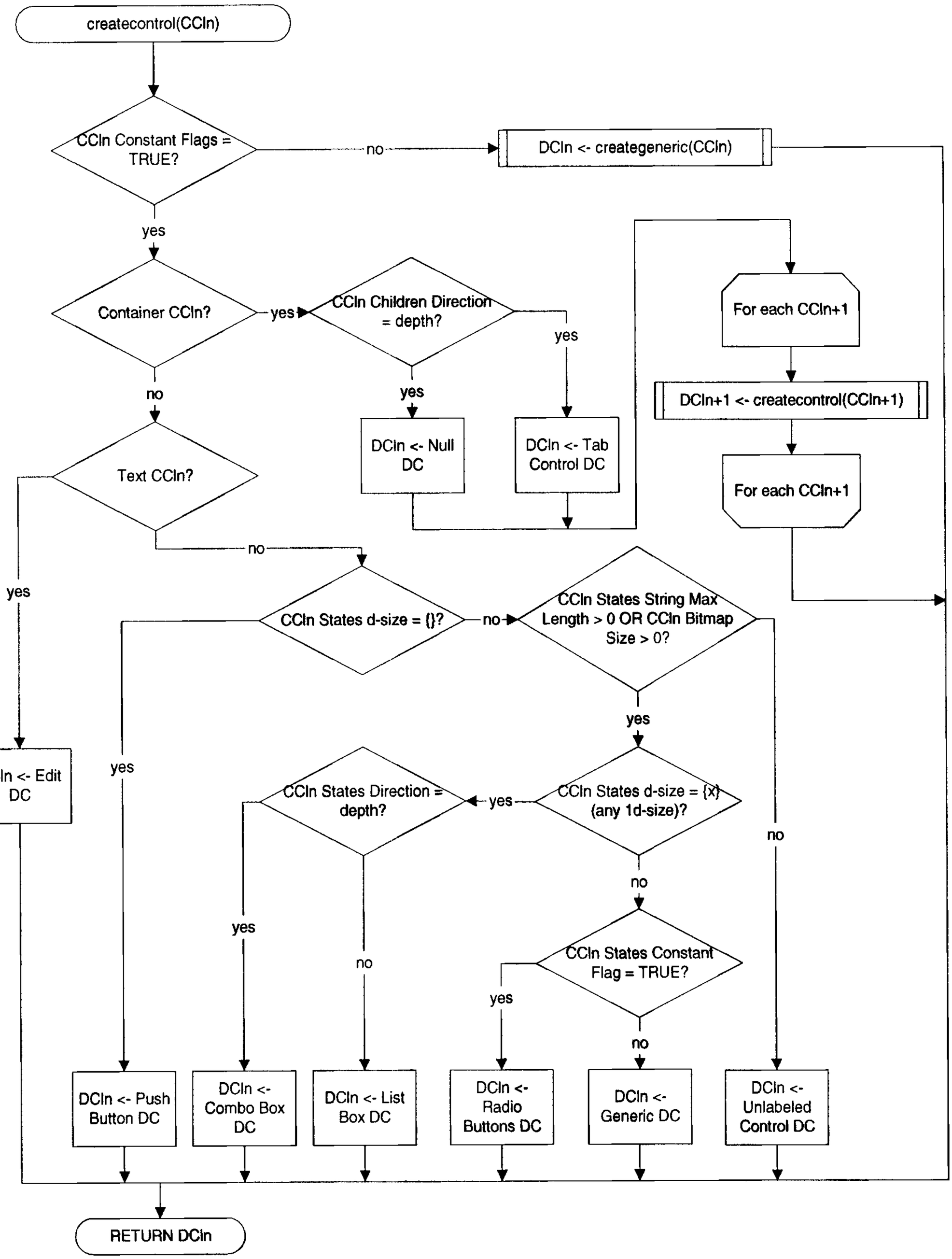

createcontrol(CCIn)
CCIn Constant Flags = TRUE?
no
DCIn <- creategeneric(CCIn)
yes
Container CCIn?
yes
CCIn Children Direction = depth?
yes
no
yes
DCIn <- Null DC
DCIn <- Tab Control DC
Text CCIn?
no
yes
CCIn States d-size = {}?
no
CCIn States String Max Length > 0 OR CCIn Bitmap Size > 0?
yes
DCIn <- Edit DC
yes
CCIn States Direction = depth?
yes
CCIn States d-size = {x} (any 1d-size)?
no
yes
no
yes
CCIn States Constant Flag = TRUE?
yes
no
DCIn <- Push Button DC
DCIn <- Combo Box DC
DCIn <- List Box DC
DCIn <- Radio Buttons DC
DCIn <- Generic DC
DCIn <- Unlabeled Control DC
RETURN DCIn
For each CCIn+1
DCIn+1 <- createcontrol(CCIn+1)
For each CCIn+1

FIG. 18G

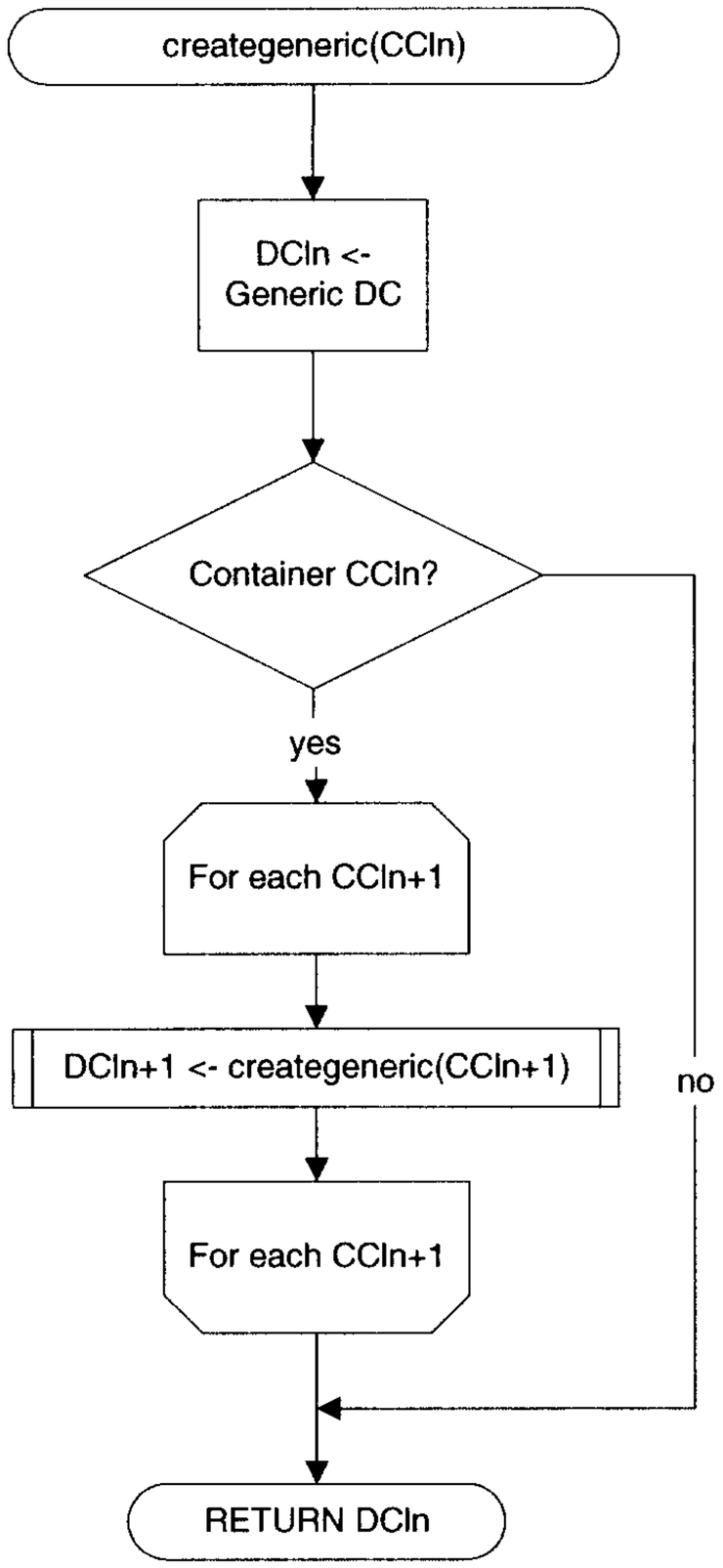


FIG. 18H

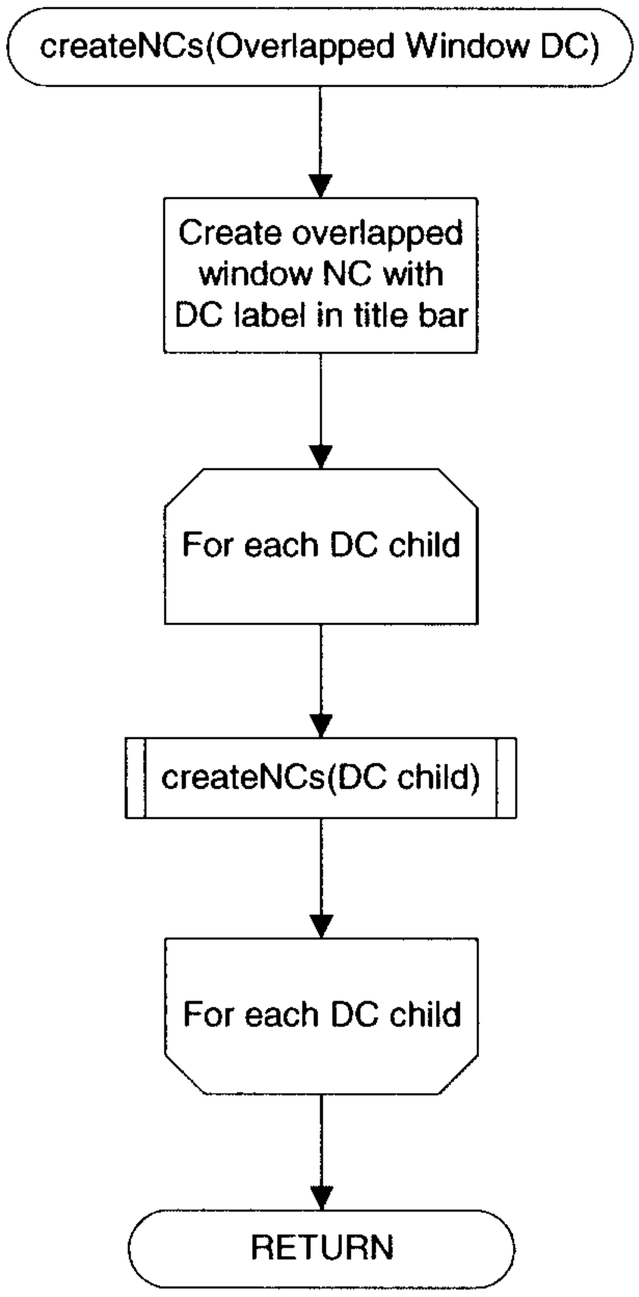


FIG. 19A

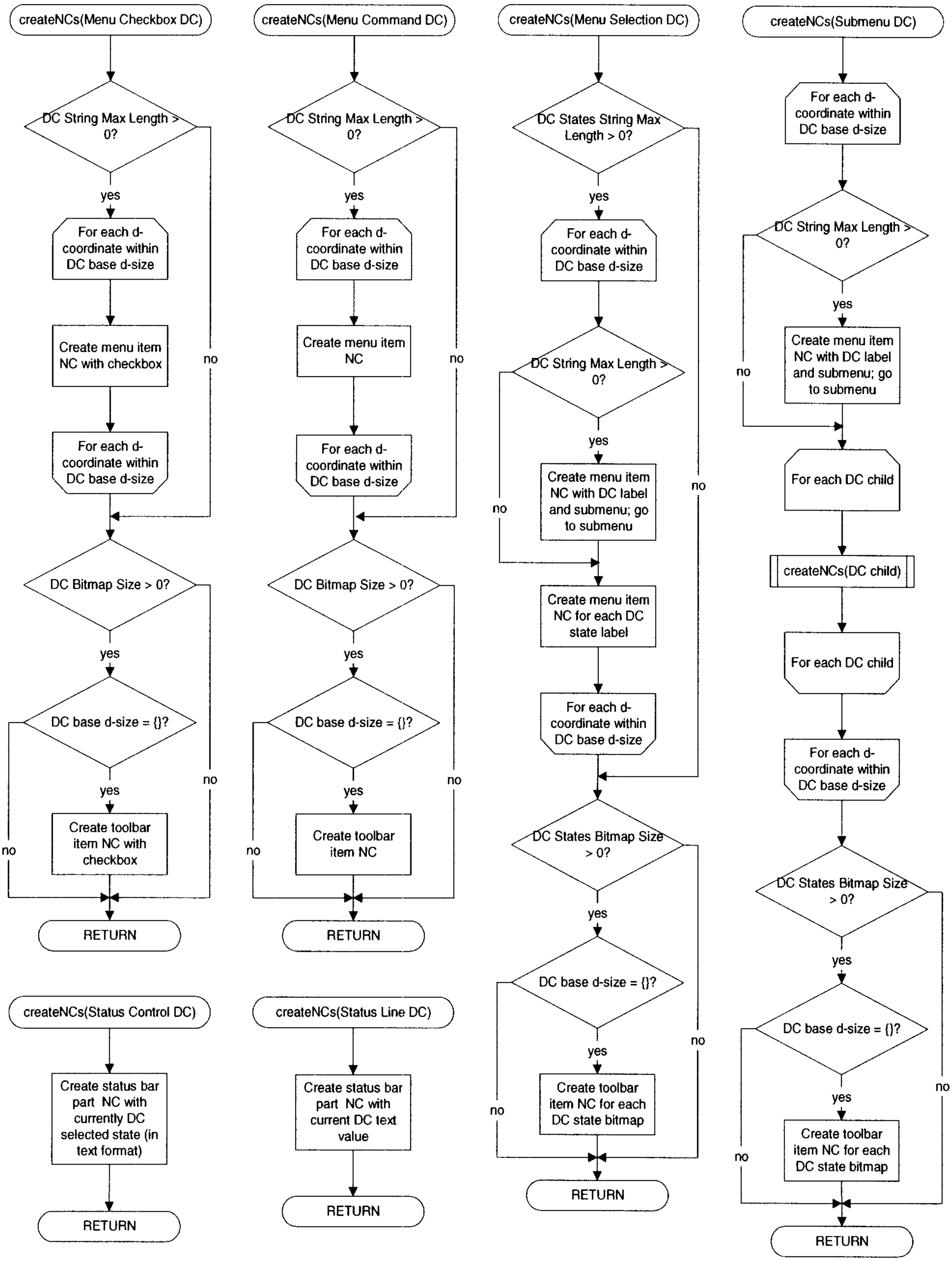

createNCs(Menu Checkbox DC)
DC String Max Length > 0?
yes
no
For each d-coordinate within DC base d-size
Create menu item NC with checkbox
For each d-coordinate within DC base d-size
DC Bitmap Size > 0?
yes
no
DC base d-size = {}?
yes
no
Create toolbar item NC with checkbox
RETURN
createNCs(Menu Command DC)
DC String Max Length > 0?
yes
no
For each d-coordinate within DC base d-size
Create menu item NC
For each d-coordinate within DC base d-size
DC Bitmap Size > 0?
yes
no
DC base d-size = {}?
yes
no
Create toolbar item NC
RETURN
createNCs(Menu Selection DC)
DC States String Max Length > 0?
yes
no
For each d-coordinate within DC base d-size
DC String Max Length > 0?
yes
no
Create menu item NC with DC label and submenu; go to submenu
Create menu item NC for each DC state label
For each d-coordinate within DC base d-size
DC States Bitmap Size > 0?
yes
no
DC base d-size = {}?
yes
no
Create toolbar item NC for each DC state bitmap
RETURN
createNCs(Submenu DC)
For each d-coordinate within DC base d-size
DC String Max Length > 0?
yes
no
Create menu item NC with DC label and submenu; go to submenu
For each DC child
createNCs(DC child)
For each DC child
For each d-coordinate within DC base d-size
DC States Bitmap Size > 0?
yes
no
DC base d-size = {}?
yes
no
Create toolbar item NC for each DC state bitmap
RETURN
createNCs(Status Control DC)
Create status bar part NC with currently DC selected state (in text format)
RETURN
createNCs(Status Line DC)
Create status bar part NC with current DC text value
RETURN

FIG. 19B

Conceptual User Interface

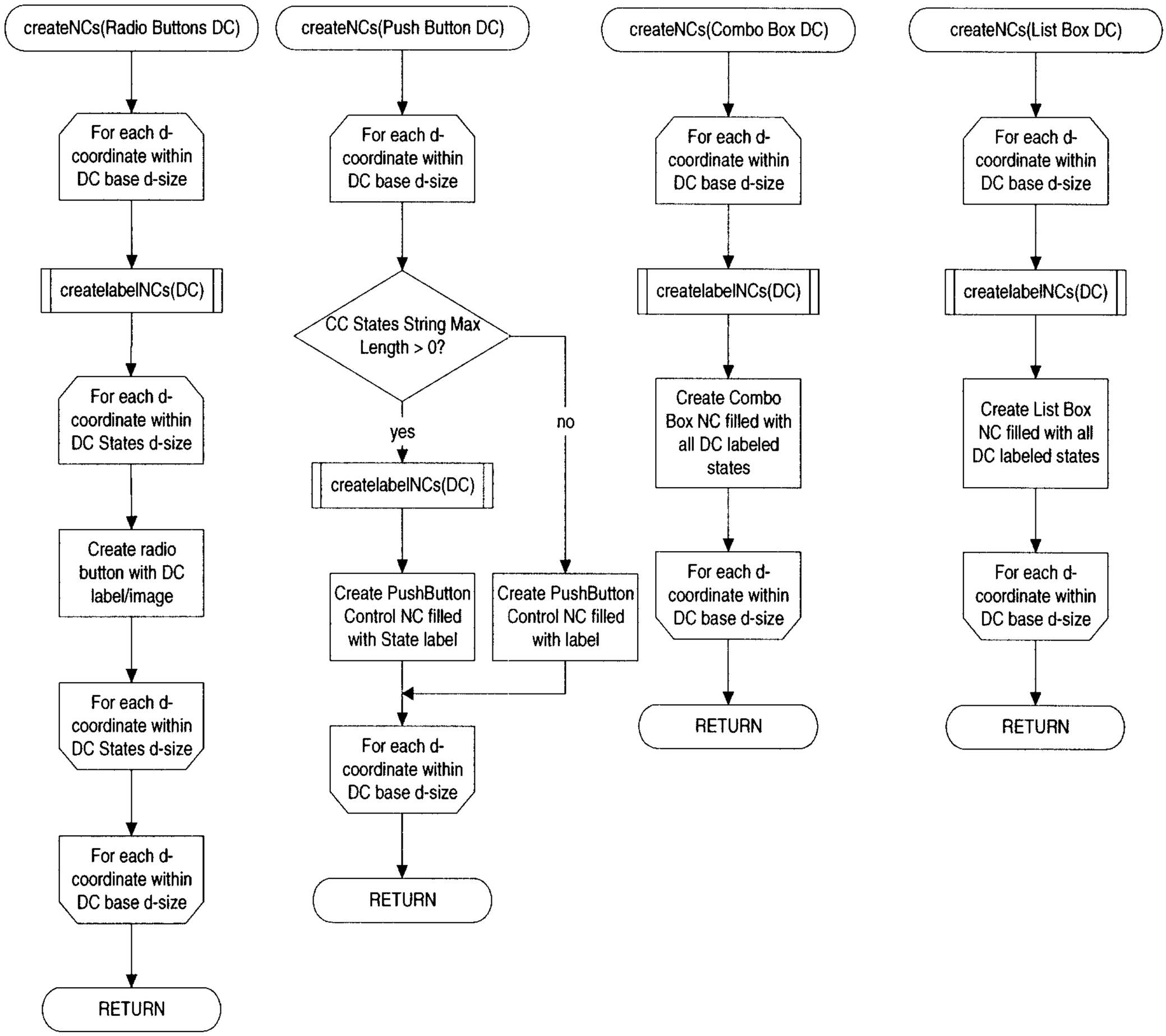


FIG. 19C

Conceptual User Interface

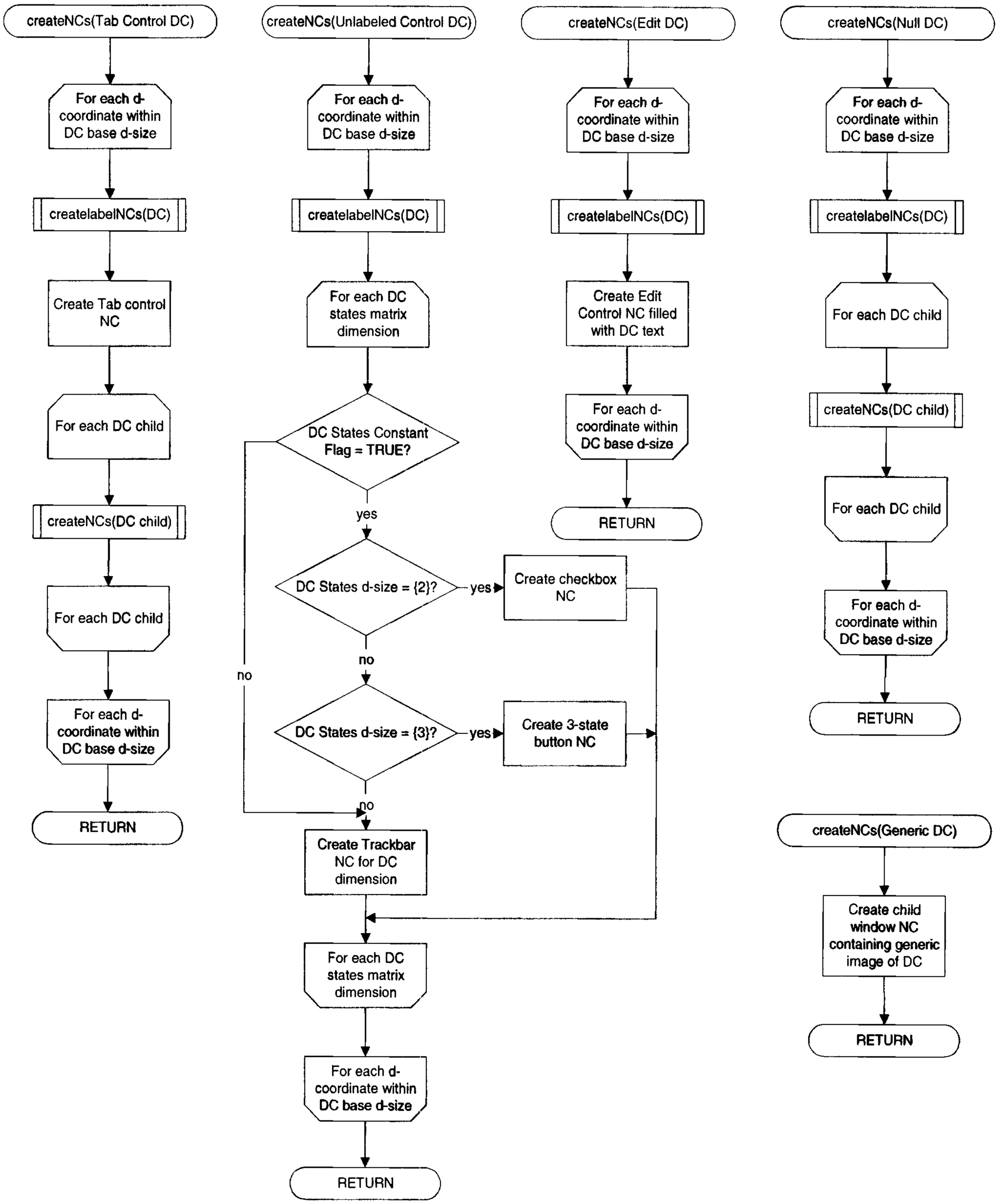


FIG. 19D

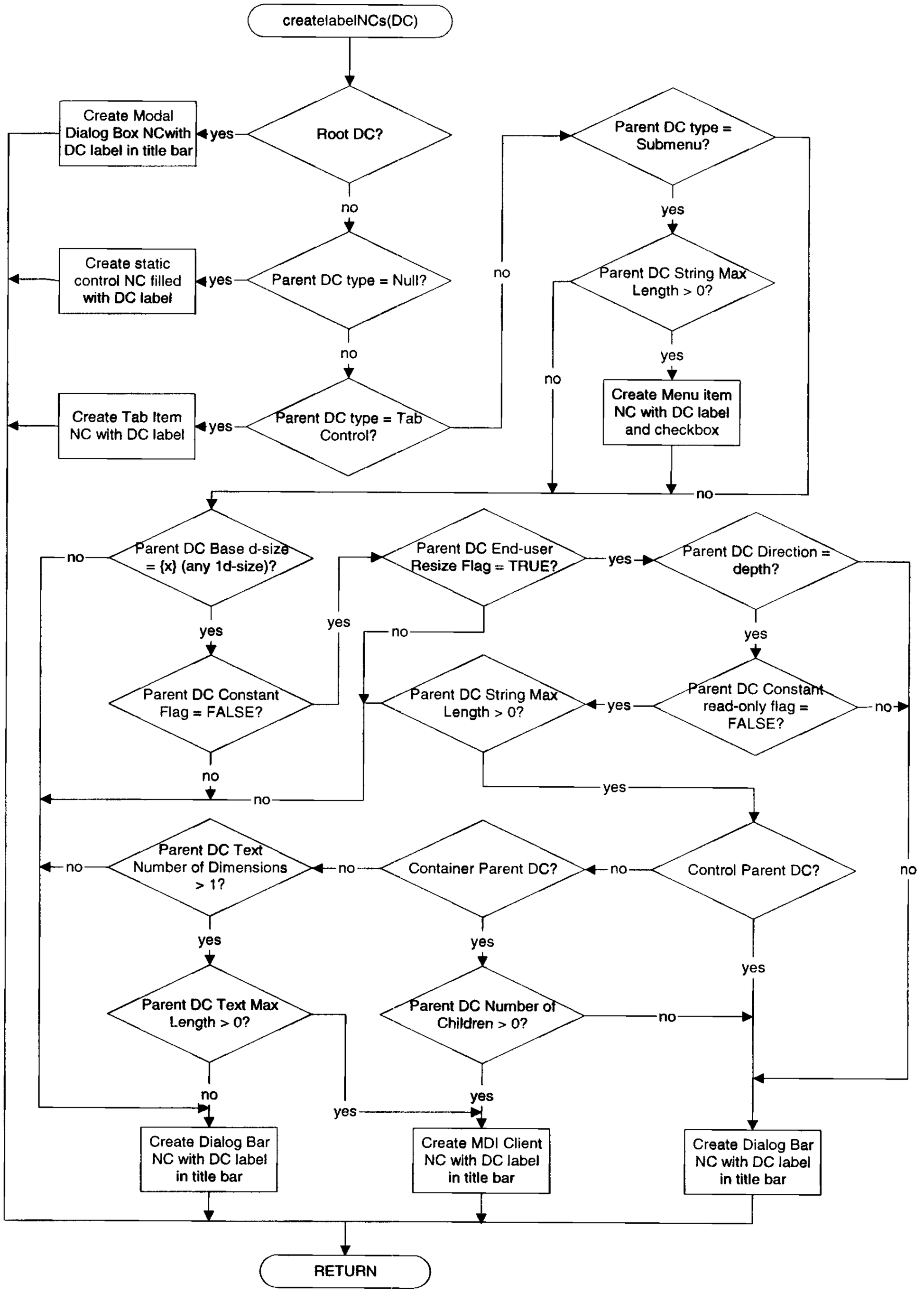

createlabelNCs(DC)
Root DC?
yes
Create Modal Dialog Box NCwith DC label in title bar
no
Parent DC type = Null?
yes
Create static control NC filled with DC label
no
Parent DC type = Tab Control?
yes
Create Tab Item NC with DC label
no
Parent DC type = Submenu?
yes
Parent DC String Max Length > 0?
yes
Create Menu item NC with DC label and checkbox
no
no
Parent DC Base d-size = {x} (any 1d-size)?
no
yes
Parent DC Constant Flag = FALSE?
yes
no
no
Parent DC End-user Resize Flag = TRUE?
yes
no
Parent DC String Max Length > 0?
yes
Parent DC Direction = depth?
yes
Parent DC Constant read-only flag = FALSE?
yes
no
yes
Parent DC Text Number of Dimensions > 1?
no
yes
Parent DC Text Max Length > 0?
no
yes
Container Parent DC?
no
yes
Parent DC Number of Children > 0?
no
yes
Control Parent DC?
no
yes
no
Create Dialog Bar NC with DC label in title bar
Create MDI Client NC with DC label in title bar
Create Dialog Bar NC with DC label in title bar
RETURN

FIG. 19E

Conceptual User Interface

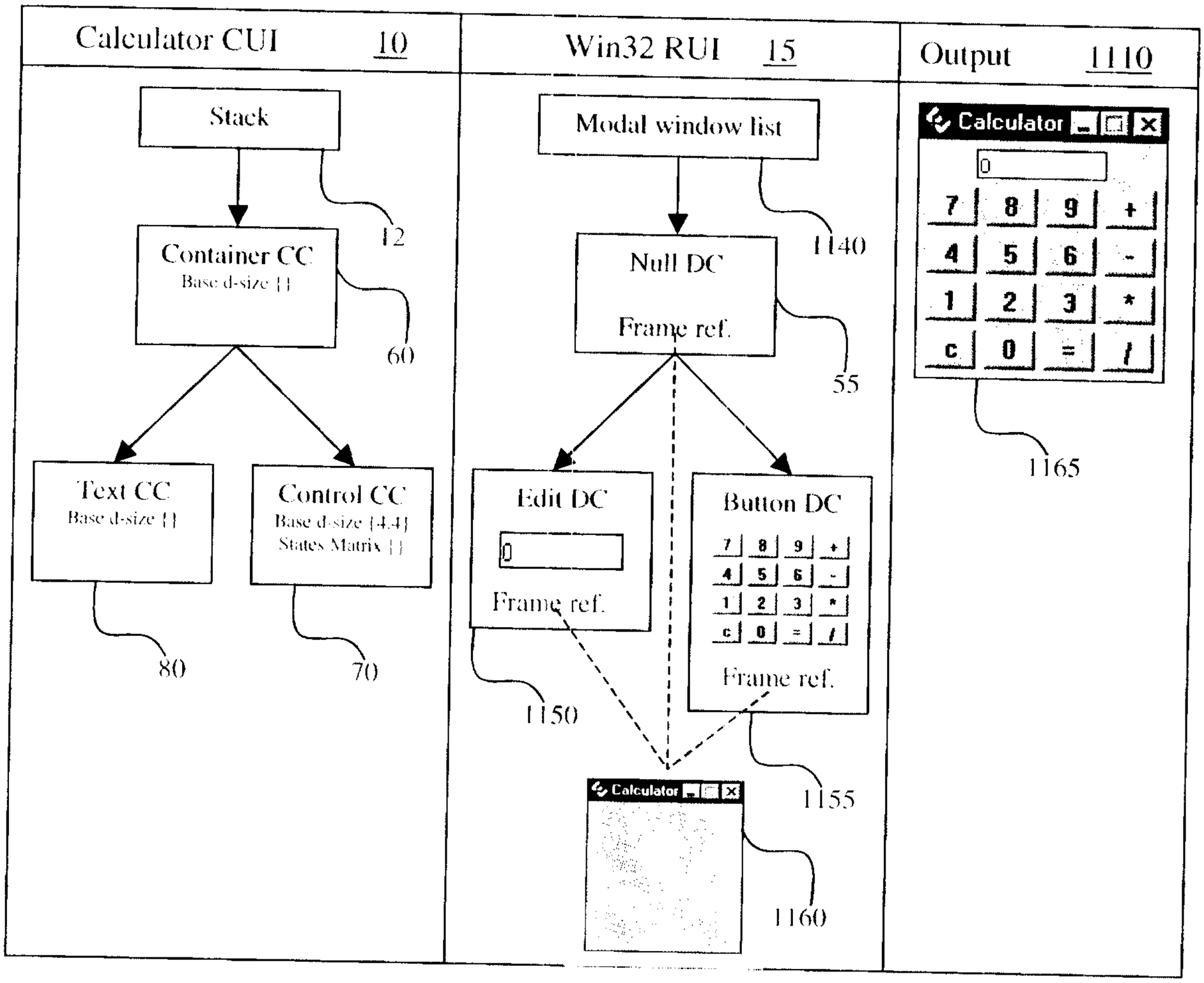
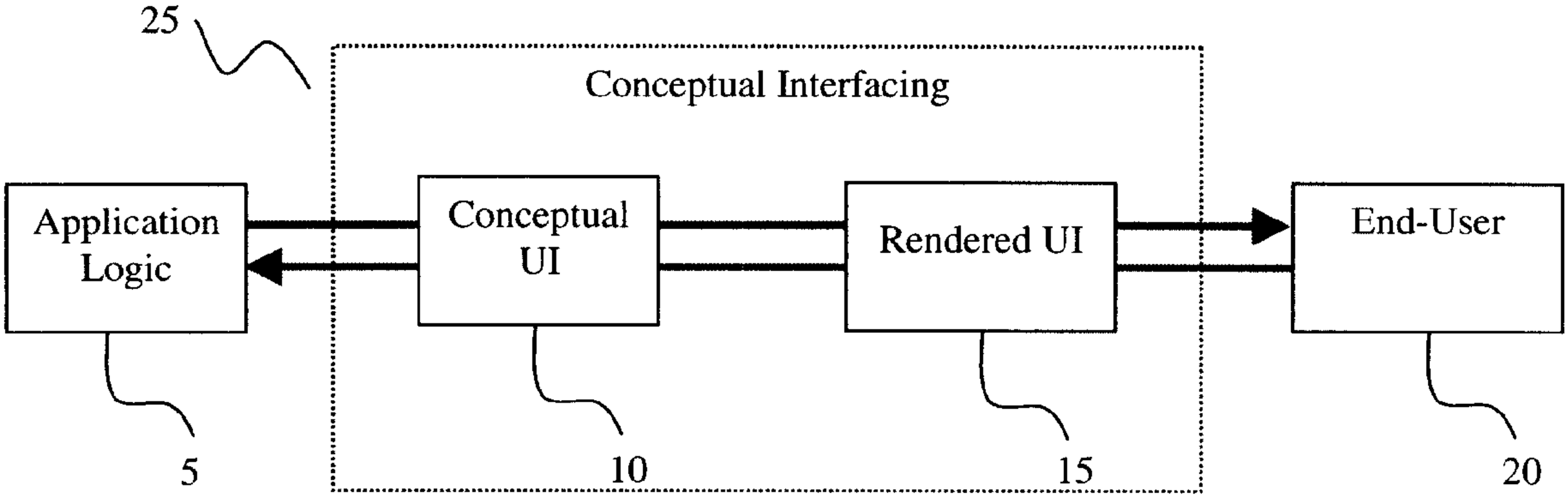


FIG. 20


25
Conceptual Interfacing
Application Logic
Conceptual UI
Rendered UI
End-User
5
10
15
20