(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 811 193 B1

(12) EUROPEAN PATENT SPECIFICATION

(45) Date of publication and mention of the grant of the patent:
14.10.1998 Bulletin 1998/42

(21) Application number: **96903932.0**

(22) Date of filing: **22.02.1996**

(51) Int Cl.[6]: **G06F 9/44**

(86) International application number:
PCT/DK96/00080

(87) International publication number:
WO 96/26484 (29.08.1996 Gazette 1996/39)

(54) GRAPHICAL ENVIRONMENT FOR MANAGING AND DEVELOPING APPLICATIONS

GRAPHISCHE ENTWICKLUNGS- UND VERWALTUNGSUMGEBUNG FÜR ANWENDUNGSPROGRAMME

ENVIRONNEMENT GRAPHIQUE DE GESTION ET DE DEVELOPPEMENT D'APPLICATIONS

(84) Designated Contracting States:
DE DK FR GB IE IT SE

(30) Priority: **22.02.1995 US 392164**

(43) Date of publication of application:
10.12.1997 Bulletin 1997/50

(73) Proprietor: **Egilsson, Agust S.**
Madison, WI 53719 (US)

(72) Inventor: **Egilsson, Agust S.**
Madison, WI 53719 (US)

(74) Representative:
Plougmann, Vingtoft & Partners A/S
Sankt Annae Plads 11,
P.O. Box 3007
1021 Copenhagen K (DK)

(56) References cited:
EP-A- 0 597 316 **US-A- 5 255 363**

- **CHI '92 CONFERENCE PROCEEDINGS, 3 - 7 May 1992, MONTEREY, pages 499-506, XP000426829 RAYMONDE GUINDON: "Requirements and Design of DesignVision, an Object-Oriented Graphical Interface to an Intelligent Software Design Assistant"**

EP 0 811 193 B1

Description

Technical Field

In general the present invention relates to an application building environment. More specifically, the present invention relates to an application building and managing environment in which users of computer systems, including ones with minimal programming knowledge, create and share powerful software applications.

Background of the Invention

In a conventional application building environment such as in most third and fourth generation programming languages, the task of constructing an application requires intensive knowledge of specialized methods and structures. Existing application environments do not enable end-users with limited programming knowledge to combine program modules from other users and program developers into complicated applications. The sharing of program components is done on a level inaccessible by the end-user. Furthermore, existing systems do not combine programming at different skill and complication levels into a unified environment where complex programming and simple end-user programming follow the same approach. The components of an application are usually rigorously embedded into the program source code either in the form of text or as a combination of text and objects. This results in difficulties in sharing computer components. Module and data dictionaries and well organized program libraries provide a partial solution to this problem. However there are still very limited possibilities for end-users of computer systems to share or even build complicated applications involving rich modular programming structure without intensive knowledge of a complicated programming environment.

The introduction of multi-dimensional spreadsheets allows users of such systems to create more complicated calculation models through a smooth transaction from two-dimensional spreadsheets and is thus a step toward making applications manageable by the end-user. Current spreadsheet systems however do not focus the user's attention on the underlying logical modular structure ever-present in an application. Furthermore current spreadsheet systems do not generalize the spreadsheet concept in such a way that powerful programming techniques can be applied without leaving the methodology of the spreadsheet environment but instead employ add-on techniques such as the writing of macros to enhance usability.

There exist numerous source code generators and fourth generation programming environments using graphical structures as part of application generation. Source code generators however and most other application building environments, by definition, are intended for program development and provide an environment for developing applications which is entirely different from the end-user environment. Thus the end-user is separated from the application building environment and is unable to create or share application components without moving up to the application developing level.

U.S. Patent No. 4,956,773, issued Sep.11,1990 to Saito et al., describes methods for creating programs using various diagrams.

U.S. Patent No. 5,255,363, issued Oct. 19, 1993 to Seyler, describes methods for writing programs within a spreadsheet application.

U.S. Patent No. 5,317,686, issued May 31, 1994 to Salas et al., describes re-labelling of spreadsheet cells.

Conference proceeding paper: CHI'92 CONFERENCE PROCEEDINGS, 3-7 May 1992, Monterey, pages 499-506, XP000426829; R.GUINDON: "Requirements of design of DesignVision, an object-oriented graphical interface to an intelligent software design assistant". The paper describes methods for viewing and editing modules at different levels in the calling hierarchy associated with a given module, it also describes dividing a program module into input, output, and other parts.

European Patent Application No. EP-A-0 597 316, by Joseph, Eugene R. et al., describes methods for programming through pictorial means and discusses dividing a program module into different parts. The Joseph et al application also defines a State Table that is sometimes referred to in the application as being spreadsheet-like.

Summary of the Invention

The object of the current invention is to allow a program of an advanced functional language and in particular an indication of the value of output variables of the program to be represented in a cell format as in a spreadsheet. This object is achieved by the method of claim 1 and the system of claim 11.

A preferred embodiment of the present invention provides a graphical application developing and managing environment in which the building and sharing of application components is done largely by the end-user.

This object is accomplished by combining a graphical logical modular structure diagram, methods for embedding source code for a class of programming languages into a visual spreadsheet-like format and standardized interfaces into modules and external applications.

Modular structure is present in any software application and can be visualized in many different ways depending on which program parts are grouped together to form a module and how they are interconnected. However the term logical modular structure used above represents a modular structure describing a solution to a specific task an application is intended to perform.

In the preferred embodiment application or module building is done by drawing on a display unit a logical

modular structure diagram describing how the new module is connected to other modules representing existing applications, previously written modules and unfinished modules. Using this diagram detailed description about the application is attached to the various parts of the diagram so as to represent the flow of information between the modules and the manipulation and visual representation of information. The new module is assigned an interface part. The interface part is a standardized interface, through which future applications communicate, and which can be used as an interface to external applications such as databases. The embodiment thus enables the creation of standardized interfaces, visually represented, into modules including modules which overlay external applications. The embodiment provides methods for using as an equivalent alternative to the text format a spreadsheet-like format when writing and executing source code written in a functional programming language, methods for moving back and forth from the spreadsheet format and the traditional text format and methods for attaching source code to the parts of the above logical modular structure diagram. The term functional programming language or a functional language refers to a programming language in which each statement written in the language can be evaluated, based on its dependent variables, to a uniquely defined value of some type. This does not automatically exclude languages in which order of execution is important, since we do not require the languages to be "pure functional languages", see for example in the literature: Sethi, Ravi. Programming Languages, Addison-Wesley Publishing Company, 1990. The term advanced functional language applies to a functional language which can be used to handle and return more complex variables than the basic variable types text and number. An example of a very simple source code (module Audio) written according to the above principle is shown in connection with the description for **Fig.8** of the drawings.

Detailed Description of the Invention

Normal spreadsheet systems do not extend to allow the general representation of code written in an advanced functional language as discussed in the present description.

In important embodiments, the system includes means for displaying the full value of a variable associated to a cell in the cell format. These means may be provided by a program communicating to the user additional information about the object referred to by the value associated to the cell and possibly submitting the value to another system capable of providing insight into the value when such a system is available. Displaying the full value thus attempts to display a value in a system native to the value. An example of a full value would be to open up a table frame to display the content of a table referred to in a cell. This is done, depending on the embodiment, by submitting the table value to a system capable of displaying and possibly allowing the editing of the content of the table in a table frame or including a procedure capable of displaying the content of the table in the embodiment itself. Another example is submitting a sound variable, referred to in a cell, to a sound system thus allowing the user to play or edit the sound within the system.

In any practical embodiment, the system should include means for editing the statement defining the variable associated to a cell in the cell format. These means may, e.g., be provided by a program implementing a text editor displaying the definition of a statement associated to a cell, upon request, and allowing the user to edit the statement definition similar to the editing of functions and formulas in most spreadsheet applications. These means are at a lower level connecting an input device and a process device in a way which enables the editing.

In some cases it may be convenient for the user to be able to view some part of the program specifications in a pure text format and the invention provides means for doing that and for going between the two formats, i. e., the cell and the standard text format. These means may be provided by using the standard text format and by implementing a translation algorithm, see, e.g., **Fig. 10** and **Fig.11**, to move back and forth between the display formats.

In an embodiment of the invention, the cell format is used together with additional means to structure and visualize the writing and editing of a program application further. A system suitable for this embodiment would further include means comprising:

(a) Means for communicating with an input device connected to the process device for allowing the editing of text and figures.

(h) Means, provided, e.g., by a program capable of drawing parent-child relational diagrams, for displaying a modular structure diagram on the display device representing the call declarations part of the program specification.

(i) Means for organizing the remaining program specification, excluding the call declarations, of said program specification into predetermined smaller parts each containing a selection of variables and their definitions. These means may be provided by a program, usually part of the parsing mechanism and possibly accepting user input, to classify the statements in said program specification.

(j) Means for displaying at least some of said parts and, if desired, each of said parts, on the display device. These means may be provided by displaying the statements in each part either in text or cell format.

(k) Means for editing or creating the program specification by editing at least some of said parts and/or by editing said modular structure diagram. The editing of the statements in text or cell format may

be done by a text editor as described above. Editing said modular structure diagram is usually done with a graphical editor capable of representing parent-child modular relationships, and the resulting modular structure is translated to conform to the programming language used.

In most programming languages, code can be divided into several parts which are expressed differently in different languages, but serve the same purpose. Such a division is obtained by classifying program specifications into input declarations, output declarations, call declarations, call definitions and program statements. For this division it is assumed that said input declarations, when assigned values, define values of said output declarations using said output declarations, said call declarations, said call definitions and said program statements. It is also assumed that said call definitions define a flow of information through the modular structure existing in the program and that the modular structure is described by said call declarations. An embodiment of the invention uses this classification and represents each of the parts, not necessarily independently, in various formats for viewing, execution, tracing and editing.

Logically this classification is related to the modular structure diagram and, in an embodiment, an association is created between areas of the diagram, for example knots representing modules and lines representing the flow between modules, by assigning at least some of the program parts, or if desired each of the program parts, excluding call declarations, to specific areas or objects of the diagram. The user can then access a specific part of the program specifications by using an input / pointing device to designate an object or area of the modular structure diagram. These additional means may be implemented using well-known program routines.

In a preferred embodiment there is an application building and sharing environment, comprising:

(a) A collection of, possibly user defined, program modules stored on storage devices.

(b) Display means capable of defining a work area on a display device.

(c) Module organizing means for organizing a program module from said collection into predefined parts including an interface part, call declaration part and call definition part. These means may be provided by a program, usually part of the parsing mechanism and possibly accepting and storing user input at some point, to classify the statements in said program module.

(d) Icon representation means capable of assigning at least some of the program modules, or if desired each of the program modules, from said collection icons and capable of displaying said icons on the work area. These means may be provided by associating each module to a geometrical figure or a bitmap and possibly also text and displaying the resulting figure and text on said work area.

(e) Interface representation means capable of displaying on the work area said interface part of a program module from said collection using a cell format. Means for viewing statements in the cell format are described above.

(f) Data flow representation means capable of displaying on said work area using a cell format the call definition part of a parent-child module relationship within a program module from said collection. Examples of such means are explained in connection with **Fig.7**.

(g) Structure means capable of displaying on the work area a modular structure diagram for a program module from the collection representing graphically the parent-child module relationships declaration part within the program module, see, e. g., **Fig.4**.

(h) Sharing means to allow users to select program modules from said collection and represent a selected program module by its icon on said work area. These means may, in an embodiment, be provided by a shared folder-like system allowing users to select the program module and locate it on the work area using an input / pointing device.

(i) Program module editing means capable of allowing users to edit and add program modules to said collection by editing each of said predefined parts of a program module from said collection.

(j) Program module execution means capable of determining and displaying, using the interface cell format of a program module from said collection, indication values representing results of executing said program module definitions. These means may be provided by an algorithm implementing an indication function after the execution (implies parsing and compiling or interpreting) of the code has assigned values to the relevant variables.

One embodiment of the environment is implemented as follows. The above program module editing means allow a user of the application environment to edit graphically said modular structure diagram to create and delete parent-child relationships within the module. The data flow definitions are edited within the data flow cell format, explained in connection with **Fig.7,** and wherein said interface part is edited within the interface cell format, see **Fig.5** and **Fig.6**. The graphical editing of the modular structure (call declarations) and the editing of the flow between modules using the data flow cell format defines a patchwork mechanism in the environment. Thus, it becomes possible for the user to define program and flow variables and view their values simultaneously using a patchwork mechanism.

For a user not allowed to change the definition of a particular module in the collection, this will be sufficient

to allow the user to view results obtained by executing the module. In order to be able to define the module, using the cell format, the environment must not exclude the remaining statements (the program part) from being edited in the cell format.

In an application environment, in which users have access to shared programming modules, there should be means to control the access of the individuals to the modules. In an embodiment, these means may be provided by associating to each module information about the privileges required to use the module and comparing it to privileges of users trying to access the module.

In a preferred embodiment of the environment it is important to be able to adjust to at least some other existing programming environments. This may be achieved by implementing a compiler, for the programming language used, returning object code compatible to other systems, also referred to here as exporting a module to another system.

The above application building and sharing environment may further include means for associating to and displaying with the module icon, of a program module from above collection, icons representing user selected variables within the program module specification. These means may be provided by associating a control icon, usually a button, to a specific variable in the program specification. Then means for allowing a user to access the full value of the variable, associated to the control icon, may be provided by executing, upon request, the program parts needed to define the variable and when the variable value has been determined the full value is displayed. Since, for non-constants, the value of a variable in a module is only defined when the input variables are defined, this assumes that the system has available input values for the module. A suitable way is to store for each user the input values used in the last execution of the module or a set of default input values. In a spreadsheet system this is always the case, i. e., values are not removed from the cells after execution of the program.

A preferred embodiment also incorporates a method of representing with an application (e.g. module) icon, at least some of the executable parts from within an application. A system useful for performing this method comprises:

(a) A program module, defining an application, specified using an advanced functional language.
(b) An icon representing said application.
(c) Means for displaying icons on a display device.
(d) Means for allowing a user to select variables from within said program module and represent at least some of the selected variable independently with a respective icon (such as icon **402** shown on **Fig. 4**). These means are provided by enabling the user to choose from the source code which variables should be represented individually.
(e) Means for allowing the user to define the display positions of the variable icons relative to the application icon. This may be done by storing and using the relative position, selected by the user, of the variable icon.
(f) Means for displaying the application icon and the variable icons on a display device in the configuration determined by the user. These means are usually provided by a figure and bitmap display mechanism and by retrieving the relative position of each of the variable icons from storage.
(g) Means for communicating with an input / pointing device and displaying the full value of at least some of the variables selected when the associated variable icon is designated by said input device. Similar means are discussed above and require the system to have available input values for the module.

This generates a setting in which a user of the application can execute, at run-time, each of the selected variables from within the application. In the case of a value pointing to an external document or object, this allows the user to access the (selected) external documents used in the program module.

Updating child modules in the system generates the well known problem of compatibility with existing parent modules. This issue is addressed by providing the system with means for representing graphically multiple versions of modules as explained in connection with **Fig.4**.

Thus, the system includes means for storing multiple versions of said program module and means for displaying with the application icon a time axis and time icons, means for associating to at least some of the time icons a specific version of said program module and means for displaying the icon configuration according to the version associated to a time icon designated by said input device. Using time axes in accordance with this embodiment can, of course, be generalized, beoynd the environments otherwise discussed herein, to more generalized settings in graphical environments where multiple versions of programs are used.

Brief Description of the Drawings

Fig. 1 is a schematic block diagram of a system arrangement for the present invention.

Fig. 2 is a conceptual diagram showing a program module divided into parts representing program parts defining the module; also shown are external applications used in the module definition.

Fig. 3 is a conceptual diagram showing a relationship between a parent module and child modules.

Fig. 4 shows modules and external applications represented by icons and relationships between parent and child modules represented by a modular structure diagram.

Fig. 5 explains the embedding of a program module

into a cell format.

Fig. 6 explains the embedding of a program module into a mixed text and cell format.

Fig. 7 explains the embedding of data flow specifications between parent and child modules into a cell format.

Fig. 8 is an overview diagram showing how the various parts of a program module on a text form are represented visually.

Fig. 9 is a flow chart for displaying a variable value of a statement in a functional language in a cell.

Fig. 10 is a flow chart for embedding a program source code on text format into a cell format.

Fig. 11 is a flow chart for translating source code on cell format into text format.

Fig. 12 is a flow chart for responding to a command to display the full value of a variable.

Fig. 13 is a schematic block diagram explaining multiple display formats.

Fig. 14 is a block diagram explaining the process of exporting module definitions to other systems and using module definitions with a form designer.

Fig. 15 and **Fig. 16** are pictures used in the example contained in the description associated with **Fig. 8**.

Description of Preferred Embodiments

In the following, examples of embodiments of the present invention are described.

Fig. 1 shows a typical system arrangement for performing the present invention. Numeral **101** denotes one or more CPU, **102** denotes computer memory, **103** denotes data storage such as hard disks or any other type of data storage, **105** denotes one or more input devices such as a pointing device and a keyboard, **106** denotes one or more display devices capable of displaying graphical and geometrical figures and text, **107** denotes zero or more output devices such as a printer. Numeral **104** designates one or more processors, operating systems and mechanisms connecting **101**, **102**, **103**, **104**, **105**, **106** and **107** into a computer system. This arrangement includes, but is not limited to, computers connected together by a network.

Fig. 2 illustrates a generic module. A generic module **(202)** includes an interface part **(205)**, connections to other modules **(206)**, data flow specifications between modules **(207)** and other program statements **(208)**. Interface part **205** is composed from input **(203)** and output **(204)** specifications. Specifications **203**, **204**, **206**, **207** and **208** define output **204**, sometimes called results, as a function of input **203**. The specifications **(203, 204, 206, 207** and **208)** form a program specification **(209)** defining module **202**. Statements in specification **209** can include statements referring to and using other documents or applications **(201)**.

Fig. 3 illustrates relationships **(301)** between modules. A module **(302)** of type **202**, referred to as parent module, uses results from other modules, called child modules, by specifying input values or conditions for interface part **205** of child modules. Existence of relationships **301** is specified in the connections part **206**, sometimes called call declarations, of module **302** and the flow of data between parent module **302** and child modules, sometimes called call definitions, is specified in flow part **207** of module **302**.

Fig. 4 shows an icon **(401)** representing module **302**. Icon **401** is connected to other icons related to the module such as an icon **(402)** representing an external application variable. Each child module is assigned an icon (one of which is numbered **404)** and each parent-child relationship is assigned an icon (one of which is numbered **403**, usually a curve) connecting icon **401** to a child module icon. Each icon is assigned methods capable of responding to input devices **105**. This defines a diagram **(405)** called a modular structure diagram and is displayed on a display device. Diagram **405** replaces the need for text specification for call declarations **206** for module **302**, and a user is allowed to create a graphical figure **405** instead of writing specifications **206**. To icon **401** the user has the option of associating another icon **(406)** referred to here as time axis. On icon **406**, time axis, time icons **(407)** are located representing different versions of module specifications **209** for module **302**. Which versions are represented by icons (e.g. **407)** is determined by the user(s) creating the module specifications. Versions of specification **209** for module **302** for which a time icon is placed on time axis **406** are stored along with the current version of the specification. Modules having time axis associated with them are displayed in modular structure diagrams by their icons and with or without the time axis icon depending on whether the current version of specifications is to be used or an older version is selected. The version selected is represented by a time icon **(408)** having attributes or shapes distinguishing it from other time icons on the time axis.

Fig. 5 shows program module **302** represented in a spreadsheet like format **(504)**, called cell format here, composed of cells displayed on a display device. Cell format **504** contains areas designated as input variables **(501)** also denoted by i1, i2, ..., output variables (**502**) also denoted by o1, o2,..... and manipulation variables **(503)** defining respectively input **203**, output **204** and program part **208** of module **302**. Cell format **504** defines the input, output and program part of module **302** by a functional language embedded and viewed in the cell format as described by flow chart **901** and explained below. Cell format **504** also defines a runtime environment for module **302** by allowing the user to modify definitions of cells and simultaneously view result of program operations as is standard within spreadsheet environments.

Fig. 6 shows program module **302** viewed in a mixed cell - and text format **(601)** composed of cells **501** and **502** representing input and output variables and program code **208** represented by a text area **(602)** containing a program source code **(603)**. Source code **603**

together with **501** and **502** define elements number **208**, **203** and **204** of module **302** respectively. Flow charts **1001** and **1101** together with flow chart **901** provide methods for translating between text format **602** and cell format **503**, if program source code **603** is written in a functional language. Format **601** allows users to view results of program operations and modify program definitions simultaneously through cells **501** and **502** as in format **504**. A mixed cell - and text format enables the user to choose a programming environment best suitable and enables the usage of more than one programming languages. Users select which format to work with, to define and use module **302**, by means of input devices **105** by choosing from methods attached to icon **401**. A full text format is possible and can be translated back and forth to a full cell format or mixed cell - and text format assuming that the text specification is done using a functional language.

Fig. 7 shows the data flow specifications **301** between parent module **302** and a single child module **(702)**. Data flow specifications **301** is represented in a spreadsheet like cell format **(701)** by denoting the rows in **701** by input variables **203** (denoted by i'1, i'2,...) and output variables (denoted by o'1, o'2, ...) **204** from interface part **205** of module **702**. Columns are denoted by call variables (denoted by c1, c2,...) in such a way that each column c1, c2, ... corresponds to a call to child module **702** with input values in rows i'1, i'2, ... and results shown in rows o'1, o'2, ... if the user chooses to view results at the same time as specifying data flow **301**. Values passed back and forth through relationship **301** are of a general type determined by the programming language in which program specification **209** of module **302** is specified in and the values are embedded into cell format by methods described by flow chart **901**. An alias **(**denoted by A in **701)** is assigned to the relationship between parent module **302** and child module **702**. Variables (denoted by A:c1:i'1,..., A:c2:o'2, ...) in cell format **701** are made available to specification **209** of module **302**. Each parent-child relationship of module **302** is assigned a user editable flow specification similar to **701** and in this way call definition part **207** of module **302** is replaced by a spreadsheet like environment. Methods for accessing the data flow specifications between parent modules and child modules are attached to icons **(**e.g. **403)** between the modules.

Fig. 8 provides an overview over how in the present invention each part of program specification **209** is represented graphically or in cell format replacing the need for a text specification defining **209**. Module **302** is represented by icon **401** and time axis **406** to specify version as indicated by line **801** and explained in **Fig. 4**. Call declarations **206** for module **302** are represented by modular structure diagram **405** as indicated by **802** and explained in **Fig. 4** also. Interface part **205** of module **302** is represented by cell format **501** for input declarations **203** and by cell format **502** for output declarations **204** as indicated by lines **804** and **805** respectively and explained in **Fig. 5**. Program statements **208** of module **302** are represented by, spreadsheet like, cell format **503**, as indicated by line **806**, or by text format **602** and explained in **Fig. 5** and **Fig. 6** respectively. Call definitions **207** of module **302** defining the data flow between modules are represented by sheets of cell format **701** one sheet for each relationship in modular structure diagram **405** as is explained in **Fig. 7** and indicated by line **803**. Interface part **205** of module **302** defines how other modules can access module **302** and in particular can be considered a interface into external applications **201** of module **302** or more generally a interface into a process which combines information from all modules in modular structure diagram **405** and connected external applications if present. Type definitions of variables used in input - **203**, output - **204**, call definitions **207** and program statements **208** are associated to their corresponding cells in the cell format.

In the preferred embodiment all definitions of modules, representations, users, and icons are stored in a centrally located database (in data storage **103)** accessible to all users. Users share access to modules by placing icons **(**e.g. **401)** in a folder directory structure, also stored in the database, defining access rights of other users and associating a description of the module to the icon for clarity. This enables the sharing of components between users of the system (e.g. network) having different programming skills, including users with only knowledge of spreadsheet systems, since data flow between modules can be defined using cell format **(**e.g. **701)** and a runtime environment is defined by a cell format also **(**e.g. **501** and **502)**. In order to access a particular module, the user, using input devices **105**, selects it from its folder, shown on a display device, and places it in a modular structure diagram describing a new application or activates it **(**e.g. runtime environment **504)**. In the preferred embodiment, compiling of modules is done relative to cells viewed at each moment on the display device used and all changes in underlying structures are reflected immediately. Users are warned and informed about how changes in module definitions affect other modules and have the option of using time axis to leave parent-child relationships unchanged. The usage of time axis allows existing modules to function unchanged when modules are upgraded.

Example: Here some of the features shown in Fig. 8 are explained. The following is a simple source code written in text format. The language used is classified as an "advanced functional language" by the definition used earlier. The module (Audio) searches a table (B2 or "OPERAS.DB") containing sound files and locates the sound according to a keyword (B5) in the default key for the table. The variable B3 represents the row in the table containing the sound specification (B7). The sound specification is in a column of the table with the heading "SOUND".

```
Module Audio
-Input(B5 string)
-Output(B7 sound)
Begin
  B2 := table("C:\SOUND"; "OPERAS.DB").
  B3 := row(B2; B5).
  B7 := item(B3; "SOUND").
End.
```

When viewed in cell format the input **203**, output **204** and the program statements **208** for the above source code are shown using indication values resulting from some choice (made by the user) of input values (B5 here, in the below B5 is assigned the keyword "Otello"). Evaluation of the module binds the variable B2 to a table object "OPERAS.DB" located on a storage device indexed by "C:\SOUND" in this case. The indication value for B2 is " Operas" which informs the user of the system that variable B2 points to a table named "Operas". The variable B2 is then referred to in other cells, e. g., the formula *row(B2; B5)* defines cell / variable B3, in the same way as in a normal spreadsheet application.
When the user is editing a particular cell the original definition of the variable is displayed, e.g., when editing cell B2 the text *table("C:\SOUND"; "OPERAS.DB")* is used. Similarly the variable B3 is assigned to a row of the table and has indication value " Otello", see
Fig. 15, and the resulting sound object located in the table and assigned to variable / cell B7 has indication value " Otello". An embodiment could thus represent the module using cell format **501**, **502** and **503** as shown by picture **1501** in **Fig. 15**.
When editing the cells B2, B3 and B7 the actual text definitions in the source code appear as formulas defining the values shown. When editing the input cell / variable B5 the input value chosen is displayed for editing, here "Otello".
In order to access the above Audio module from another module (Enhanced Audio **(1602)** in **Fig. 16)** the user places the audio module as a child module **(1601)** in a modular diagram **(1604)**, shown in **Fig. 16**.
Defining the flow (call definition) between the new module and the Audio module is then done by the user using cell format 701 as shown by **1603** in
Fig. 16. In the above the input variable of the Audio module (named here Audio:Selected:B5) is assigned the value "Don Giovanni" and the
resulting sound object is assigned to the variable indexed (named) by "Audio:Selected:B7" and shown using indication value " Don Giovanni". The variable "Audio:Selected:B7" is recognized in the module Enhanced Audio above and is used to refer to the sound object resulting from the call to the Audio module.

Fig. 9 shows flow chart **(901)** describing a method for displaying a variable value of a general type in a cell in such a way that extends the way mathematical formulas with number values and text is viewed in cells. A variable - or a cell value, can be considered an array holding information about the value as determined by the variable type and the variable definition. The value array is associated to the cell holding the variable/cell definition. Information in the variable/cell value array is entered into the array by a program execution mechanism which updates the value array to reflect changes in other related variables/cells or in the variable/cell definition itself. One type of the information in the array is the variable type. The program execution mechanism also triggers the display mechanism to change the displayed value of the cell. Values considered are of general types such as a number, formula, array, geometrical object, form, database, table or a row type as determined by the programming language. A variable of a text or number type is displayed in a cell by its formatted value. The same applies to some other types such as date and time. In preferred embodiments, values of more general types are displayed in cells by applying a function, called indication function here, to the value array which specifies a method, based on the value type, for determining the displayed cell value of the variable. The resulting displayed cell value is a combination of text and icons fitting into a cell. The icons are selected from an icon font and combined with the text to form the cell display. Attributes such as color can be applied to the cells in a standard way. An example of such a function is an indication function which assigns to a variable an icon, representing the variable type, and concatenates to the icon a keyword selected from the information about the variable in the variable array. A value resulting from applying the indication function to a variable is referred to as the indication value of the variable. The variable array is not displayed but is associated to the cell. As is standard with spreadsheets, a syntax error in the definition of a variable is indicated with an error flag in the cell. Flow chart **1201** describes a method for viewing a variable of a general type in another way namely by its full value.

Fig. 10 shows a flow chart **(1001)** for embedding a program source code on text format into a cell format assuming that the programming language used is a functional language. Each statement in a functional language is associated with a variable of some general type and it is therefore possible to associate each statement and its variable to a cell in a cell format as described below. The mechanism described by the flow chart reads the statements in the source code and determines **(1002** and **1003)** into which cell to put each statement. To which cell a statement belongs, can be determined by its variable name, information in a cross reference table about a previous embedding from a cell format into text format (see flow chart **1101)** and adjusted during modification ofthe text to keep track of variables, user

input and sequential mapping (that is a mapping of variables into some area containing enough cells). The mechanism shown stores separately information about the text format, not used in the cell format, such as the sequential order of the statements in the text **(1004)**. The mechanism shown also reads information about a previous cell format for the program, if one exists, and formats the cell format accordingly. Once a location for the statement in a cell and existing attributes have been determined by the above measures, the variable name for the statement and the cell name are equated and the variable type is registered in the type array associated to the cell **(1005)**. Conflicts which arise from naming of statement variables are resolved by always mapping a statement, whose variable name is the name of a cell in the cell format into the corresponding cell. Variable values are then displayed in the cell format **(1006)** as described by flow chart **901** for each cell after a compiling and parsing mechanism has been activated to determine the variable/cell value array for each cell used. The program is then edited in the cell format in a similar way as a conventional spreadsheet. When adding or modifying statements in a cell format the variable type is determined from the syntax of the statement entered but can, when more than one types are possible, be adjusted by the user.

Fig. 11 shows a flow chart **(1101)** for translating source code on cell format into text format. The mechanism described by the flow chart reads the statement associated to each cell and determines which variable name to use in the text format for the statement **(1102** and **1103)**. Which variable name to use for a cell can be determined by the cell name or by a translating table (referred to in flow charts as cell cross reference table). The translating table may contain information from a previous embedding of the source code into cell format. Statements in the source code are modified to use the variable name instead of the cell name and attributes and information about the cells are stored separately **(1104)** to allow for an embedding back into cell format. Information from a previous embedding into cell format can be used along with other methods such as sequential ordering of statements in text and using current attributes of statements in cells to place and format statements in the text format. Statements are written into the text format **(1105)** and assigned to their corresponding variables. Information about the type of a variable is contained in the value array associated to the variable cell in the cell format and variables in the text format are declared accordingly using the format specified by the (functional) language used.

Fig. 12 shows a flow chart **(1201)** describing a mechanism for displaying the full value, by definition here, of a variable/cell value. This mechanism is triggered for a cell in a cell format by the user using an input device such as a mouse. The mechanism uses a classification of variables by types to determine methods for displaying the full value of a variable value. Information contained in the variable/cell value array associated to a cell in a cell format points to and is centered around external documents for many variable types. Example of such types are table types representing database tables in database systems, graphical - and picture types, sound types representing sound specifications, types representing word processor documents written in different formats and many other variable types. For variable types centered around external/underlying documents, the full value is displayed by activating a process resulting in the document being viewed/represented and being editable in its native application on a separated but possibly overlying area, from the cell format, of the display device or on a device determined by its native application. Methods for activating a document in its native application are well known in the field and are usually restricted by the operating system and the native application itself. For other types such as text and numbers the full value is simply showing available details, determined using the value array, in a way which is not restricted, for ex. by size, by the cell format. Accordingly, since viewing the full value is restricted by the capabilities of the operating system and other applications, not all types are necessarily displayed using full value as indicated by **(1202)** in the flow chart. In the preferred embodiment, a list of methods for displaying the full value is available and searched **(1203)** and the corresponding method activated **(1204)** to display the full value of a variable/cell.

Fig. 13 shows a schematic block diagram. **Fig. 13** gives an overview of the duality between a text and cell format and explains the above methods further. Program statements written in text format are denoted by blocks numbered **1301** and **1303**. A statement written in a functional language can be considered to be an assignment y = f(x) where y is the resulting variable, x represents the statement input variables and f represents the statement as defined by the programming language. Variable y has a variable name and a type definition as shown in box **1301**. The statement f(x) is represented in box **1303**. Associated to text format **1301** is a text editing mechanism as shown. Translation mechanism described by flow charts **1001**, **1101** translate between the two formats. In cell format **1302** the variable y is assigned a cell and the variable - and cell name, also representing the cell location as is standard for spreadsheet environments, are equated and thus refer to the same variable y. The statement text f(x) is assigned to the cell and can thus be edited by allowing the user to select the cell for editing its contents as is standard with spreadsheet environments. A change in the cell contents results in a changed definition for the corresponding module. A cell in cell format **(1302)** is assigned a value array to store information including results of program execution as determined by a system **(1306)** capable of parsing and compiling module definitions **(1304)**, stored in memory **102** and in storage devices **103**, into an executable code. Module definitions **1304** are stored in mul-

tiple formats including graphical specifications (e.g. **405)** and on cell or text format. A change in a value array triggered by system **1306** triggers a change in how the cell is displayed as explained in flow chart **901** and indicated by line **1307** in the diagram. System **1306** is responsible for displaying the full value of a value array as described by flow chart **1201** and explained above. Changes in cell display are triggered by changes in module definitions **1304** which includes changes in cells being edited as indicated by line **1305**.

Fig. 14 shows how, in a preferred embodiment, module definitions are used with traditional form designing environments and other application development systems. Forms provide an alternative format for working with modules designed by using the above methods. Module definitions are exported **(1404)** by an exporting mechanism **(1403)** into a format defined by an application environment **(1405)** being exported to allowing environment **1405** to access a module as a function with input values **203** and return values **204**. Exporting mechanisms are well known in the art. In a preferred embodiment, a collection of methods for a predefined set of application environments and operating systems is available defining the export mechanism **1403**. Exporting of modules is controlled by users of the embodiment. In a preferred embodiment, a form design and implementation mechanism **(1401)** is available to users allowing users to create alternative display formats in accordance with the design guidelines specified by a predefined list of operating environments. Design mechanism **1401** is capable of accessing and using **(1402)** module definitions **1304** directly without using an export mechanism by considering modules as functions with input - and return values.

Claims

1. A method of representing a program specification, of an advanced functional language,

the advanced functional language being defined as a programming language capable of handling and returning more complex variables than the basic variable types text and number and in which each statement written in the language can be evaluated, based on its dependent variables, to a uniquely defined value,

and representing program execution in a cell format, comprising:

(a) communicating with a display device connected to a process device,

(b) accessing a cell frame containing multiple cells displayed on said display device,

(c) displaying text and icons in said cell frame,

(d) associating selected variables and their definitions in said program specification to cells in said cell frame,

(e) determining values of said selected variables by executing said program specification, and

(f) displaying in cells, associated to said selected variables, an indication value determined by an indication function representing values of said selected variables.

2. The method of claim 1, comprising displaying a full value of at least some of said selected variables.

3. The method of claim 2, comprising editing values of at least some of said selected variables by editing the full value of the variables.

4. The method of claim 1, comprising communicating with an input device connected to said process device for controlling the editing of said specification within said cell format.

5. The method of claim 1, comprising displaying on said display device said selected variables using a text format and translating selected variables back or forth from being displayed in said cell format and said text format.

6. The method of claim 1, further comprising editing and/or defining at least some of the variables of said program specifications by using a cell format to display and edit the variables.

7. The method of claim 1, further comprising editing and/or defining at least some of the data flow specifications of said program specifications by using the data flow cell format to display and edit data flow specifications between program modules.

8. The method of claim 1, further comprising:

(a) communicating with an input device connected to said process device,

(b) displaying a modular structure diagram on said display device and/or using variables in said program specification to represent the call declarations part of said program specification,

(c) organizing the remaining program specification, excluding the call declarations, of said program specification into predetermined smaller parts each containing a selection of variables and their definitions,

(d) displaying at least some of said parts on said display device using cell formats, and

(e) editing or creating said program specification by editing at least some of said parts using the associated cell formats and/or by editing said modular structure diagram.

9. The method of claim 8, wherein said program spec-

ification is organized into input declarations, output declarations, call declarations, call definitions and program statements and wherein said input declarations when assigned values define values of said output declarations using said output declarations, said call declarations, said call definitions and said program statements and wherein said call definitions define a flow of information through said modular structure diagram and/or define the variables in said program specification representing said call declarations.

10. The method of claim 9, comprising attaching at least some of the parts of said program specification, excluding call declarations, to predefined areas of said modular structure diagram and allowing a user to access and display the cell format of a specific part by designating, using an input device, an area of said modular structure diagram attached to said specific part.

11. A system for representing a program specification, of an advanced functional language,

the advanced functional language being defined as a programming language capable of handling and returning more complex variables than the basic variable types text and number and in which each statement written in the language can be evaluated, based on its dependent variables, to a uniquely defined value,
and representing program execution in a cell format, the system comprising:

(a) means for communicating with a display device connected to a process device,
(b) means for accessing a cell frame containing multiple cells displayed on said display device,
(c) means for displaying text and icons in said cell frame,
(d) means for associating selected variables and their definitions in said program specification to cells in said cell frame,
(e) means for determining values of said selected variables by executing said program specification, and
(f) means for displaying in cells, associated to said selected variables, an indication value determined by an indication function representing values of said selected variables.

12. The system of claim 11, further including means for displaying a full value of at least some of said selected variables.

13. The system of claim 12, further including means for editing values of at least some of said selected variables by editing the full value of the variables.

14. The system of claim 11, further including means for communicating with an input device connected to said process device for controlling the editing of said specification within said cell format.

15. The system of claim 11, further including means for displaying on said display device said selected variables using a text format and means for translating selected variables back or forth from being displayed in said cell format and said text format.

16. The system of claim 11, further including means for editing and/or defining at least some of the variables of said program specifications by using a cell format to display and edit the variables.

17. The system of claim 11, further including means for editing and/or defining at least some of the data flow specifications of said program specifications by using the data flow cell format to display and edit data flow specifications between program modules.

18. The system of claim 11, in which the system further includes means comprising:

(a) means for communicating with an input device connected to said process device,
(b) means for displaying a modular structure diagram on said display device and/or using variables in said program specification to represent the call declarations part of said program specification,
(c) means for organizing the remaining program specification,
excluding the call declarations, of said program specification into predetermined smaller parts each containing a selection of variables and their definitions,
(d) means for displaying at least some of said parts on said display device using cell formats, and
(e) means for editing or creating said program specification by editing at least some of said parts using the associated cell formats and/or by editing said modular structure diagram.

19. The system of claim 18, wherein said program specification is organized into input declarations, output declarations, call declarations, call definitions and program statements and wherein said input declarations when assigned values define values of said output declarations using said output declarations, said call declarations, said call definitions and said program statements and wherein said call definitions define a flow of information through said modular structure diagram and/or define the variables in said program specification representing said call declarations.

20. The system of claim 19, further including means for attaching at least some of the parts of said program specification, excluding call declarations, to predefined areas of said modular structure diagram and means for allowing a user to access and display the cell format of a specific part by designating, using an input device, an area of said modular structure diagram attached to said specific part.

Patentansprüche

1. Verfahren zur Darstellung einer Programmspezifikation einer hochentwickelten funktionalen Sprache,
wobei die hochentwickelte funktionale Sprache als eine Programmiersprache definiert ist, die in der Lage ist,
komplexere Variablen als die Variablengrundtypen Text und Zahl zu bearbeiten und auszugeben, und bei der jede Anweisung, die in der Sprache geschrieben ist, basierend auf ihren abhängigen Variablen, zu einem einzigen definierten Wert ausgewertet werden kann,
und zur Darstellung der Programmausführung in einem Zellenformat, wobei das Verfahren die folgenden Schritte aufweist:

(a) Kommunizieren mit einer Anzeigeeinrichtung, die an eine Datenverarbeitungseinrichtung angeschlossen ist,
(b) Zugreifen auf einen Zellenrahmen, welcher eine Vielzahl von Zellen enthält, die auf der Anzeigeeinrichtung angezeigt werden,
(c) Anzeigen von Text und Ikon-Symbolen in dem Zellrahmen,
(d) Verbinden der ausgewählten Variablen und deren Definitionen in der Programmspezifikation zu Zellen in dem Zellenrahmen,
(e) Bestimmen von Werten der ausgewählten Variablen durch Ausführen der Programmspezifikation, und
(f) Anzeigen eines Anzeigewertes in Zellen, welche zu den ausgewählten Variablen gehören, wobei der Anzeigewert durch eine Anzeigefunktion bestimmt wird und die Werte der ausgewählten Variablen darstellt.

2. Verfahren nach Anspruch 1, bei dem ein voller Wert von wenigstens einigen der ausgewählten Variablen angezeigt wird.

3. Verfahren nach Anspruch 2, bei dem die Werte von wenigstens einigen der ausgewählten Variablen editiert werden, indem man den vollen Wert der Variablen editiert.

4. Verfahren nach Anspruch 1, bei dem mit einer Eingabeeinrichtung, die an die Verarbeitungseinrichtung angeschlossen ist, für die Steuerung der Editierung der Spezifikation innerhalb des Zellenformats kommuniziert wird.

5. Verfahren nach Anspruch 1, bei dem auf der Anzeigeeinrichtung die ausgewählten Variablen angezeigt werden, indem ein Textformat verwendet wird und die ausgewählten Variablen hin und her zwischen ihrer Anzeige in dem Zellenformat und in dem Textformat übersetzt werden.

6. Verfahren nach Anspruch 1, bei dem ferner wenigstens einige der Variablen der Programmspezifikationen editiert und/oder definiert werden, indem ein Zellenformat zur Anzeige und Editierung der Variablen verwendet wird.

7. Verfahren nach Anspruch 1, bei dem ferner wenigstens einige der Datenflußspezifikationen der Programmspezifikationen editiert und/oder definiert werden, indem das Datenflußzellenformat zur Anzeige und Editierung der Datenflußspezifikationen zwischen den Programmodulen verwendet wird.

8. Verfahren nach Anspruch 1, mit den folgenden weiteren Schritten:

(a) Kommunizieren mit einer Eingabeeinrichtung, die an die Verarbeitungseinrichtung angeschlossen ist,
(b) Anzeigen eines modularen Strukturdiagramms auf der Anzeigeeinrichtung und/oder Verwenden von Variablen in der Programmspezifikation zur Darstellung des Aufrufdeklarationsteils der Programmspezifikation,
(c) Organisieren der verbleibenden Programmspezifikation mit Ausnahme der Aufrufdeklarationen, der Programmspezifikation zu vorbestimmten kleineren Teilen, die jeweils eine Auswahl von Variablen und deren Definitionen enthalten,
(d) Anzeigen von wenigstens einigen dieser Teilen auf der Anzeigeeinrichtung unter Verwendung der Zellenformate, und
(e) Editieren oder Erzeugen der Programmspezifikation durch Editierung von wenigstens einigen der Teile unter Verwendung der zugehörigen Zellenformate und/oder durch Editieren des modularen Strukturdiagramms.

9. Verfahren nach Anspruch 8, bei dem die Programmspezifikation zu Eingabedeklarationen, Ausgabedeklarationen, Aufrufdeklarationen, Aufrufdefinitionen und Programmanweisungen organisiert wird, wobei die Eingabedeklarationen bei zugewiesenen Werten Werte der Ausgabedeklarationen unter Verwendung der Ausgabedeklarationen, der

Aufrufdeklarationen, der Aufrufdefinitionen und der Programmanweisungen definieren, und wobei die Aufrufdefinitionen einen Informationsfluß durch das modulare Strukturdiagramm und/oder die Variablen in der Programmspezifikation, welche die Aufrufdeklarationen darstellen, definieren.

10. Verfahren nach Anspruch 9, bei dem wenigstens einige der Teile der Programmspezifikation mit Ausnahme der Aufrufdeklarationen an vorbestimmte Bereiche des modularen Strukturdiagramms angebracht werden und einem Benutzer den Zugriff und die Anzeige des Zellenformats eines spezifischen Teils erlauben, indem unter Verwendung einer Eingabeeinrichtung ein Bereich des modularen Strukturdiagramms, welcher an den spezifischen Teil angebracht ist, bestimmt wird.

11. System zur Darstellung einer Programmspezifikation einer hochentwickelten funktionalen Sprache, wobei
die hochentwickelte funktionale Sprache als eine Programmiersprache definiert ist, die in der Lage ist, komplexere Variablen als die Variablengrundtypen Text und Zahl zu bearbeiten und auszugeben, und bei der jede Anweisung,
die in der Sprache geschrieben ist, basierend auf deren abhängigen Variablen zu einem einzigen definierten Wert ausgewertet werden kann,
und zur Darstellung der Programmausführung in einem Zellenformat, wobei das System aufweist:

(a) eine Einrichtung zur Kommunikation mit einer Anzeigeeinrichtung, die an eine Verarbeitungseinrichtung angeschlossen ist,
(b) eine Einrichtung für den Zugriff auf einen Zellenrahmen, welcher eine Vielzahl von Zellen enthält, die auf der Anzeigeeinrichtung angezeigt werden,
(c) eine Einrichtung zur Anzeige von Text und Ikon-Symbolen in dem Zellenrahmen,
(d) eine Einrichtung für die Verknüpfung der ausgewählten Variablen und deren Definitionen in der Programmspezifikation zu Zellen in dem Zellenrahmen,
(e) eine Einrichtung zur Bestimmung der Werte der ausgewählten Variablen durch Ausführung der Programmspezifikation, und
(f) eine Einrichtung zur Anzeige in Zellen, die zu den ausgewählten Variablen gehören, eines Anzeigewertes, welcher durch eine Anzeigefunktion bestimmt wird, welche Werte der ausgewählten Variablen darstellt.

12. System nach Anspruch 11, welches ferner eine Einrichtung zur Darstellung eines vollen Werts von wenigstens einigen der ausgewählten Variablen enthält.

13. System nach Anspruch 12, welches ferner eine Einrichtung zur Editierung von Werten von wenigstens einigen der ausgewählten Variablen durch Editierung des vollen Werts der Variablen enthält.

14. System nach Anspruch 11, welches ferner eine Einrichtung zur Kommunikation mit einer Eingabeeinrichtung enthält, die an die Verarbeitungseinrichtung zur Steuerung des Editierens der Spezifikation innerhalb des Zellenformats angeschlossen ist.

15. System nach Anspruch 11, welches ferner eine Einrichtung zur Anzeige der ausgewählten Variablen auf der Anzeigeeinrichtung unter Verwendung eines Textformats enthält und eine Einrichtung zur Übersetzung der ausgewählten Variablen hin und zurück von der Anzeige in dem Zellenformat und in dem Textformat.

16. System nach Anspruch 11, welches ferner eine Einrichtung zur Editierung und/oder Definierung von wenigstens einigen der Variablen der Programmspezifikationen enthält unter Verwendung eines Zellenformats zur Anzeige und Editierung der Variablen.

17. System nach Anspruch 11, welches ferner eine Einrichtung für die Editierung und/oder Definierung von wenigstens einigen der Datenflußspezifikationen der Programmspezifikationen unter Verwendung des Datenflußzellenformats zur Anzeige und Editierung der Datenflußspezifikationen zwischen den Programmodulen enthält.

18. System nach Anspruch 11, bei dem das System ferner folgende Einrichtungen aufweist:

(a) eine Einrichtung für die Kommunikation mit einer Eingabeeinrichtung, die an die Verarbeitungseinrichtung angeschlossen ist,
(b) eine Einrichtung für die Anzeige eines modularen Strukturdiagramms auf der Anzeigeeinrichtung und/oder zur Darstellung des Aufrufdeklarationsteils der Programmspezifikation unter Verwendung der Variablen in der Programmspezifikation,
(c) eine Einrichtung zur Organisation der verbleibenden Programmspezifikation, mit Ausnahme der Aufrufdeklarationen, der Programmspezifikation zu vorbestimmten kleineren Teilen, die jeweils eine Auswahl der Variablen und deren Definitionen enthalten,
(d) eine Einrichtung zur Anzeige von wenigstens einigen der Teile auf der Anzeigeeinrichtung unter Verwendung der Zellenformate, und
(e) eine Einrichtung zur Editierung und Erzeugung der Programmspezifikation durch Editierung von wenigstens einigen der Teile unter

Verwendung der zugehörigen Zellenformate und/oder durch Editierung des modularen Strukturdiagramms.

19. System nach Anspruch 18, bei dem die Programmspezifikation zu Eingabedeklarationen, Ausgabedeklarationen, Aufrufdeklarationen, Aufrufdefinitionen und Programmanweisungen organisiert ist und bei dem die Eingabedeklarationen bei zugewiesenen Werten Werte der Ausgabedeklarationen unter Verwendung der Ausgabedeklarationen, der Aufrufdeklarationen, der Aufrufdefinitionen und der Programmanweisungen definieren und bei dem die Aufrufdefinitionen einen Informationsfluß durch das modulare Strukturdiagramm definieren und/oder die Variablen in der Programmspezifikation, welche die Aufrufdeklarationen darstellen, definieren.

20. System nach Anspruch 19, welches ferner eine Einrichtung zum Anbringen von wenigstens einigen der Teile der Programmspezifikation, mit Ausnahme der Aufrufdeklarationen, an vorbestimmte Bereiche des modularen Strukturdiagramms und eine Einrichtung enthält, die es einem Benutzer ermöglicht, auf das Zellenformat eines spezifischen Teils zuzugreifen und diesen anzuzeigen, indem er unter Verwendung einer Eingabeeinrichtung einen Bereich des modularen Strukturdiagramms, das an den spezifischen Teil angebracht ist, bestimmt.

Revendications

1. Un procédé de représentation d'une spécification de programme, d'un langage fonctionnel avancé, le langage fonctionnel avancé étant défini comme un langage de programmation susceptible de traiter et de renvoyer des variables plus complexes que les variables de base des types texte et nombre, et dans lequel chaque instruction écrite dans le langage peut être évaluée, sur la base de ses variables dépendantes, pour former une valeur spécialement définie,
et représentant une exécution de programme dans un format de cellules, comprenant les étapes consistant à:

(a) communiquer avec un dispositif d'affichage connecté à un dispositif de traitement,
(b) accéder à un bloc de cellules qui contient de multiples cellules affichées sur ledit dispositif d'affichage,
(c) afficher un texte et des icônes dans ledit bloc de cellules,
(d) associer, à des cellules dudit bloc de cellules, des variables sélectionnées et leurs définitions dans ladite spécification de programme,
(e) déterminer des valeurs desdites variables sélectionnées en exécutant ladite spécification de programme, et
(f) afficher dans des cellules, associées auxdites variables sélectionnées, une valeur d'indication déterminée par une fonction d'indication qui représente des valeurs desdites variables sélectionnées.

2. Le procédé selon la revendication 1, qui comprend un affichage d'une valeur complète d'au moins certaines desdites variables sélectionnées.

3. Le procédé selon la revendication 2, qui comprend une édition de valeurs d'au moins certaines desdites variables sélectionnées en éditant la valeur complète des variables.

4. Le procédé selon la revendication 1, qui comprend une communication avec un dispositif d'entrée connecté audit dispositif de traitement afin de commander l'édition de ladite spécification à l'intérieur dudit format de cellules.

5. Le procédé selon la revendication 1, qui comprend les étapes consistant à afficher sur ledit dispositif d'affichage lesdites variables sélectionnées en utilisant un format de texte et à traduire, dans un sens ou dans l'autre, des variables sélectionnées de manière à les afficher, soit dans ledit format de cellules, soit dans ledit format de texte.

6. Le procédé selon la revendication 1, qui comprend en outre une édition et/ou une définition d'au moins certaines des variables desdites spécifications de programme en utilisant un format de cellules pour afficher et éditer les variables.

7. Le procédé selon la revendication 1, qui comprend une édition et/ou une définition d'au moins certaines des spécifications de flux de données desdites spécifications de programme en utilisant le format de cellules de flux de données pour afficher et éditer des spécifications de flux de données entre modules du programme.

8. Le procédé selon la revendication 1, qui comprend en outre les étapes consistant à:

(a) communiquer avec un dispositif d'entrée connecté audit dispositif de traitement,
(b) afficher un schéma de structure modulaire sur ledit dispositif d'affichage et/ou utiliser des variables de ladite spécification de programme pour représenter la partie de déclarations d'appels de ladite spécification de programme ;
(c) organiser la spécification restante du programme, à l'exclusion des déclarations d'appels, de ladite spécification de programme en

parties plus petites prédéterminées, qui contiennent chacune une sélection de variables et leurs définitions,
(d) afficher au moins certaines desdites parties sur ledit dispositif d'affichage en utilisant des formats de cellules, et
(a) (e) éditer ou créer ladite spécification de programme en éditant au moins certaines desdites parties en utilisant les formats associés de cellules et/ou en éditant ledit schéma de structure modulaire.

9. Le procédé selon la revendication 8, dans lequel ladite spécification de programme est organisée en déclarations d'entrées, déclarations de sorties, déclarations d'appels, définitions d'appels et assertions de programme, et dans lequel lesdites déclarations d'entrées définissent, lorsque des valeurs leurs sont assignées, des valeurs desdites déclarations de sortie en utilisant lesdites déclarations de sortie, lesdites déclarations d'appels, lesdites définitions d'appels et lesdites assertions de programme, et dans lequel lesdites définitions d'appels définissent un flux d'information à travers ledit schéma de structure modulaire et/ou définissent les variables de ladite spécification de programme représentant lesdites déclarations d'appels.

10. Le procédé selon la revendication 9, qui comprend l'étape consistant à attacher au moins certaines des parties de ladite spécification de programme, à l'exclusion de déclarations d'appels, à des zones prédéfinies dudit schéma de structure modulaire et à permettre à un utilisateur d'accéder au format de cellules d'une partie spécifique et de l'afficher en désignant, en utilisant un dispositif d'entrée, une zone du schéma de structure modulaire attachée à ladite partie spécifique.

11. Un système de représentation d'une spécification de programme, d'un langage fonctionnel avancé,
le langage fonctionnel avancé étant défini comme un langage de programmation susceptible de traiter et de renvoyer des variables plus complexes que les variables de base des types texte et nombre, et dans lequel chaque instruction écrite dans le langage peut être évaluée, sur la base de ses variables dépendantes, pour former une valeur spécialement définie,
et représentant une exécution de programme dans un format de cellules, le système comprenant :

(a) un moyen de communication avec un dispositif d'affichage connecté à un dispositif de traitement,
(b) un moyen d'accès à un bloc de cellules qui contient de multiples cellules affichées sur ledit dispositif d'affichage,
(c) un moyen d'affichage d'un texte et d'icônes dans ledit bloc de cellules,
(d) un moyen d'association, à des cellules dudit bloc de cellules, de variables sélectionnées et de leurs définitions dans ladite spécification de programme,
(e) un moyen de détermination de valeurs desdites variables sélectionnées en exécutant ladite spécification de programme, et
(f) un moyen d'affichage dans des cellules, associées auxdites variables sélectionnées, d'une valeur d'indication déterminée par une fonction d'indication qui représente des valeurs desdites variables sélectionnées.

12. Le système selon la revendication 11, qui comprend un moyen d'affichage d'une valeur complète d'au moins certaines desdites variables sélectionnées.

13. Le système selon la revendication 12, qui comprend un moyen d'édition de valeurs d'au moins certaines desdites variables sélectionnées en éditant la valeur complète des variables.

14. Le système selon la revendication 11, qui comprend un moyen de communication avec un dispositif d'entrée connecté audit dispositif de traitement afin de commander l'édition de ladite spécification à l'intérieur dudit format de cellules.

15. Le système selon la revendication 11, qui comprend un moyen destiné à afficher sur ledit dispositif d'affichage lesdites variables sélectionnées en utilisant un format de texte et un moyen destiné à traduire, dans un sens ou dans l'autre, des variables sélectionnées de manière à les afficher, soit dans ledit format de cellules, soit dans ledit format de texte.

16. Le système selon la revendication 11, qui comprend en outre un moyen d'édition et/ou de définition d'au moins certaines des variables desdites spécifications de programme en utilisant un format de cellules pour afficher et éditer les variables.

17. Le système selon la revendication 11, qui comprend un moyen d'édition et/ou de définition d'au moins certaines des spécifications de flux de données desdites spécifications de programme en utilisant le format de cellules de flux de données pour afficher et éditer des spécifications de flux de données entre modules du programme.

18. Le système selon la revendication 11, dans lequel le système inclut en outre des moyens qui comprennent :

(a) un moyen de communication avec un dispositif d'entrée connecté audit dispositif de trai-

tement,
(b) un moyen destiné à afficher un schéma de structure modulaire sur ledit dispositif d'affichage, et/ou à utiliser des variables de ladite spécification de programme pour représenter la partie de déclarations d'appels de ladite spécification de programme ;
(c) un moyen d'organisation de la spécification restante du programme, à l'exclusion des déclarations d'appels, de ladite spécification de programme en parties plus petites prédéterminées, qui contiennent chacune une sélection de variables et leurs définitions,
(d) un moyen d'affichage d'au moins certaines desdites parties sur ledit dispositif d'affichage en utilisant des formats de cellules, et
(e) un moyen d'édition ou de création de ladite spécification de programme en éditant au moins certaines desdites parties en utilisant les formats associés de cellules et/ou en éditant ledit schéma de structure modulaire.

19. Le système selon la revendication 18, dans lequel ladite spécification de programme est organisée en déclarations d'entrées, déclarations de sorties, déclarations d'appels, définitions d'appels et assertions de programme, et dans lequel lesdites déclarations d'entrées définissent, lorsque des valeurs leurs sont assignées, des valeurs desdites déclarations de sortie en utilisant lesdites déclarations de sortie, lesdites déclarations d'appels, lesdites définitions d'appels et lesdites assertions de programme, et dans lequel lesdites définitions d'appels définissent un flux d'information à travers ledit schéma de structure modulaire et/ou définissent les variables de ladite spécification de programme représentant lesdites déclarations d'appels.

20. Le système selon la revendication 19, qui inclut en outre un moyen destiné à attacher au moins certaines des parties de ladite spécification de programme, à l'exclusion de déclarations d'appels, à des zones prédéfinies dudit schéma de structure modulaire et un moyen destiné à permettre à un utilisateur d'accéder au format de cellules d'une partie spécifique et de l'afficher en désignant, en utilisant un dispositif d'entrée, une zone du schéma de structure modulaire attachée à ladite partie spécifique.

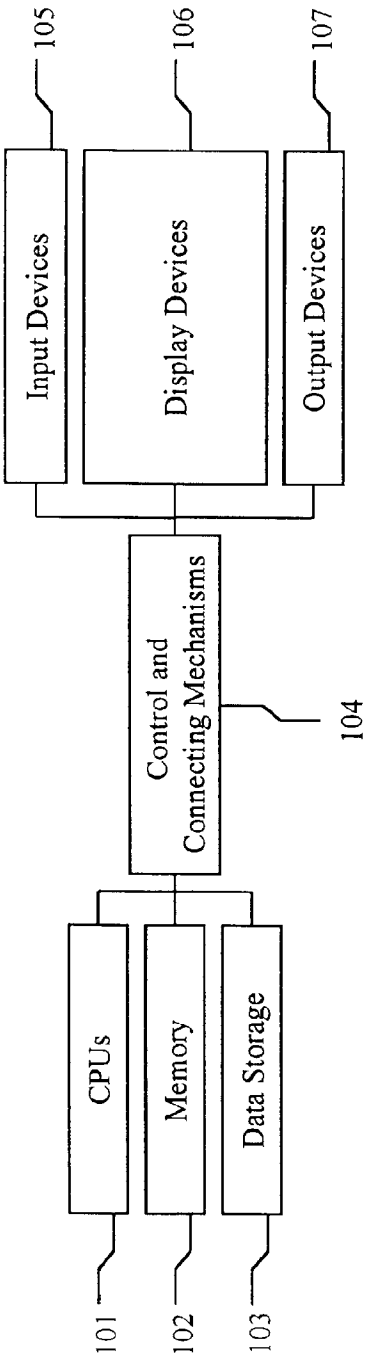
105
106
107
Input Devices
Display Devices
Output Devices
Control and Connecting Mechanisms
104
CPUs
Memory
Data Storage
101
102
103

Fig. 1

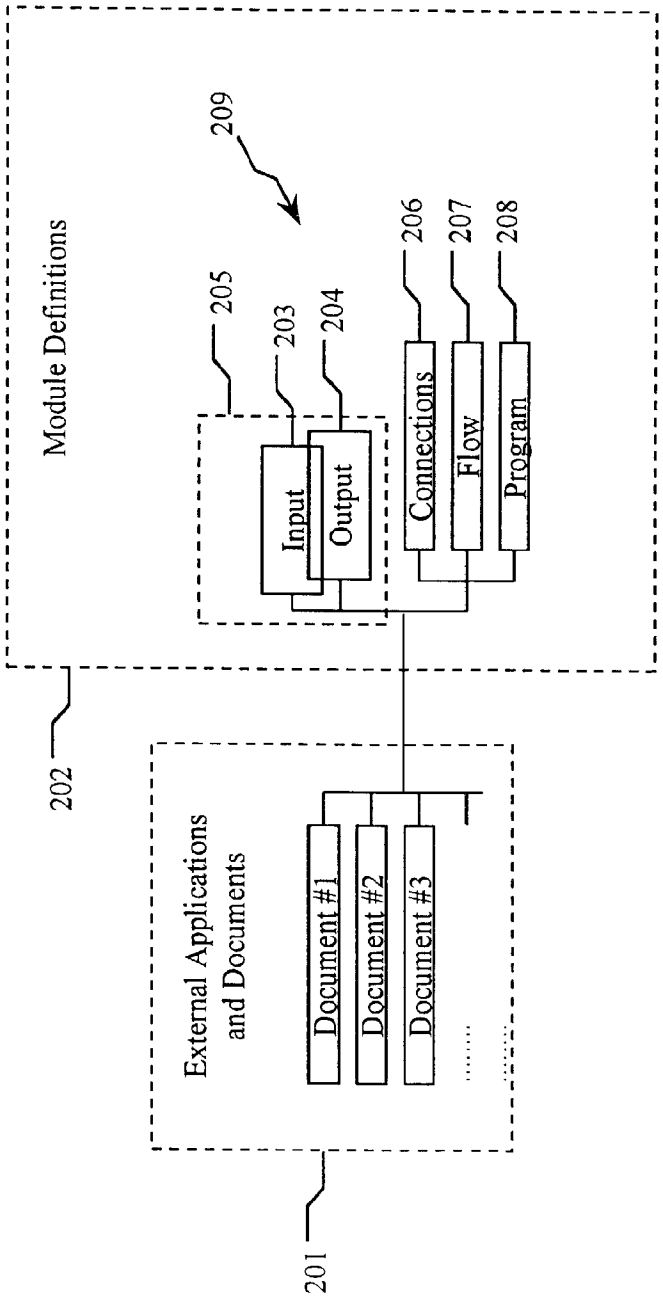
Module Definitions
209
205
203
204
206
207
208
Input
Output
Connections
Flow
Program
202
External Applications
and Documents
Document #1
Document #2
Document #3
201

Fig. 2

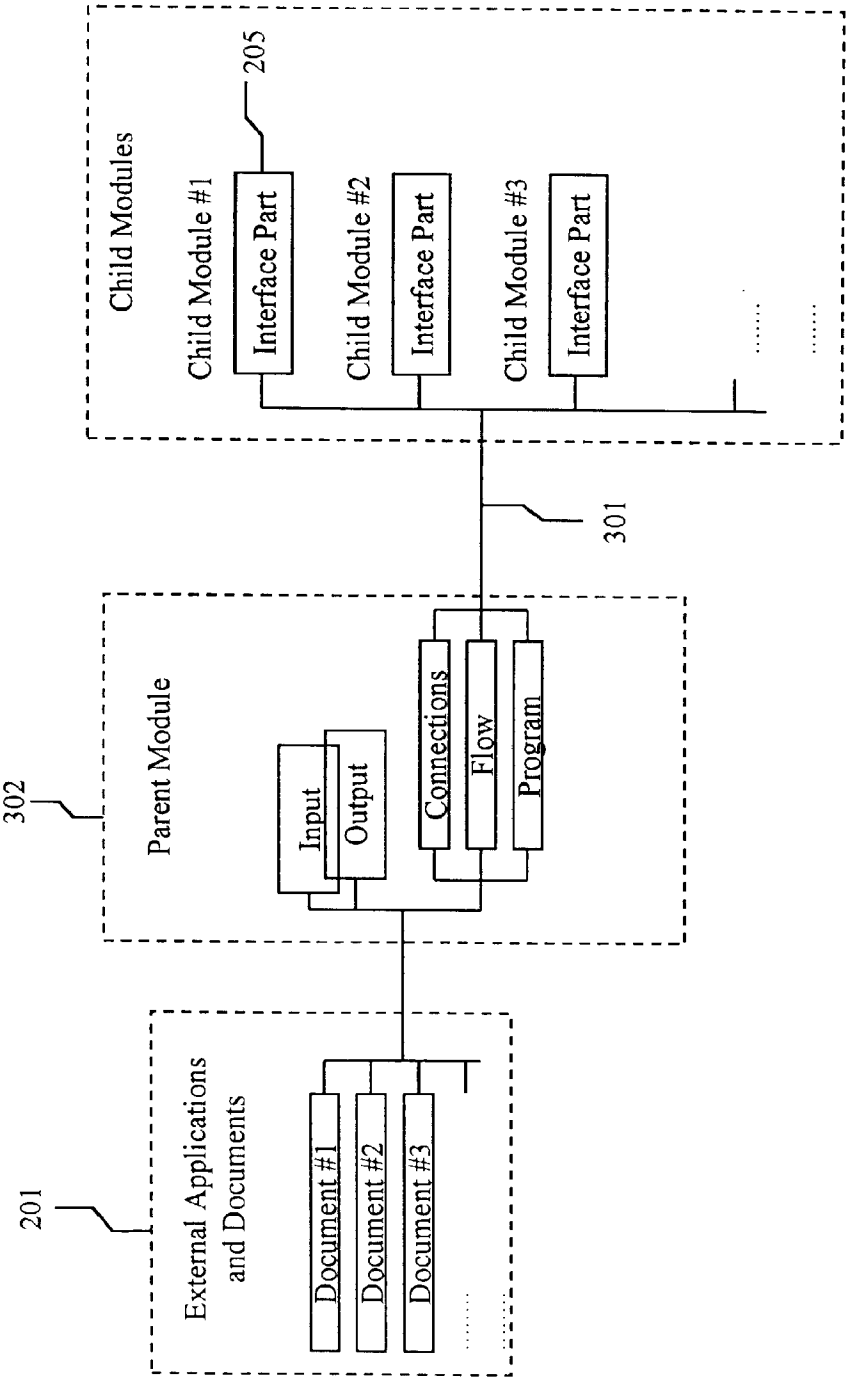
Child Modules
205
Child Module #1
Interface Part
Child Module #2
Interface Part
Child Module #3
Interface Part
301
302
Parent Module
Input
Output
Connections
Flow
Program
201
External Applications and Documents
Document #1
Document #2
Document #3

Fig. 3

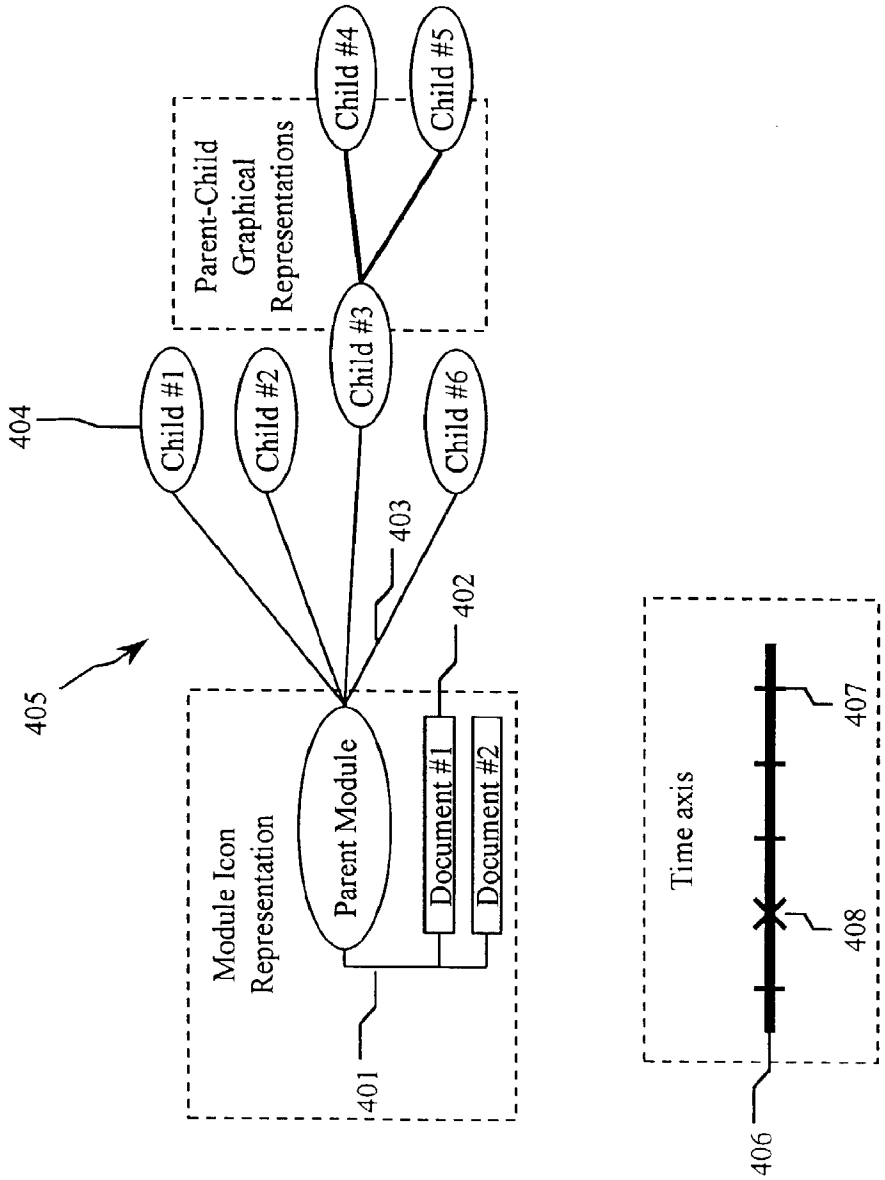
Module Icon Representation
Parent Module
Document #1
Document #2
Parent-Child Graphical Representations
Child #1
Child #2
Child #3
Child #4
Child #5
Child #6
Time axis
401
402
403
404
405
406
407
408

Fig. 4

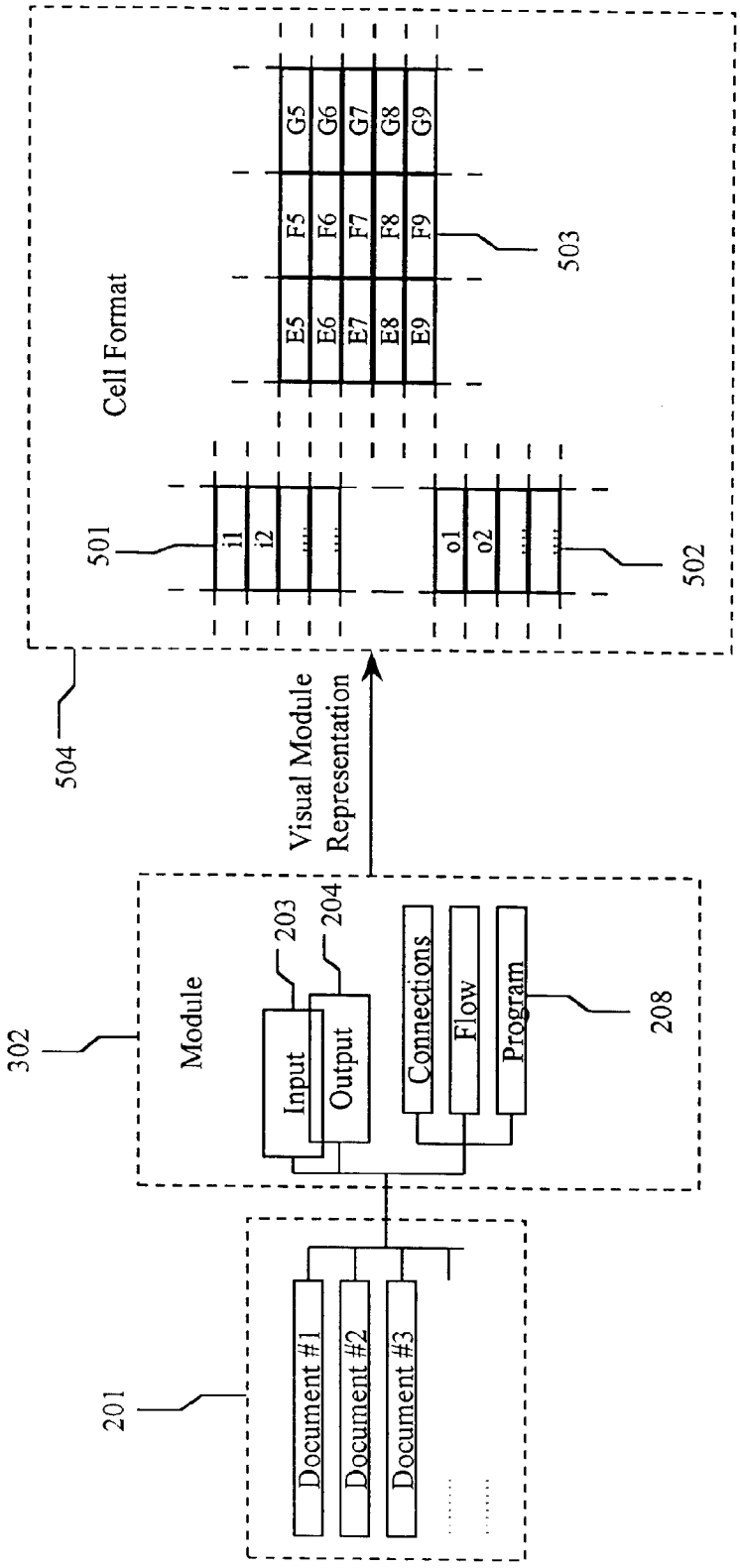
Document #1
Document #2
Document #3
201
Module
302
Input
Output
203
204
Connections
Flow
Program
208
Visual Module Representation
504
Cell Format
501
i1
i2
o1
o2
502
E5
E6
E7
E8
E9
F5
F6
F7
F8
F9
G5
G6
G7
G8
G9
503

Fig. 5

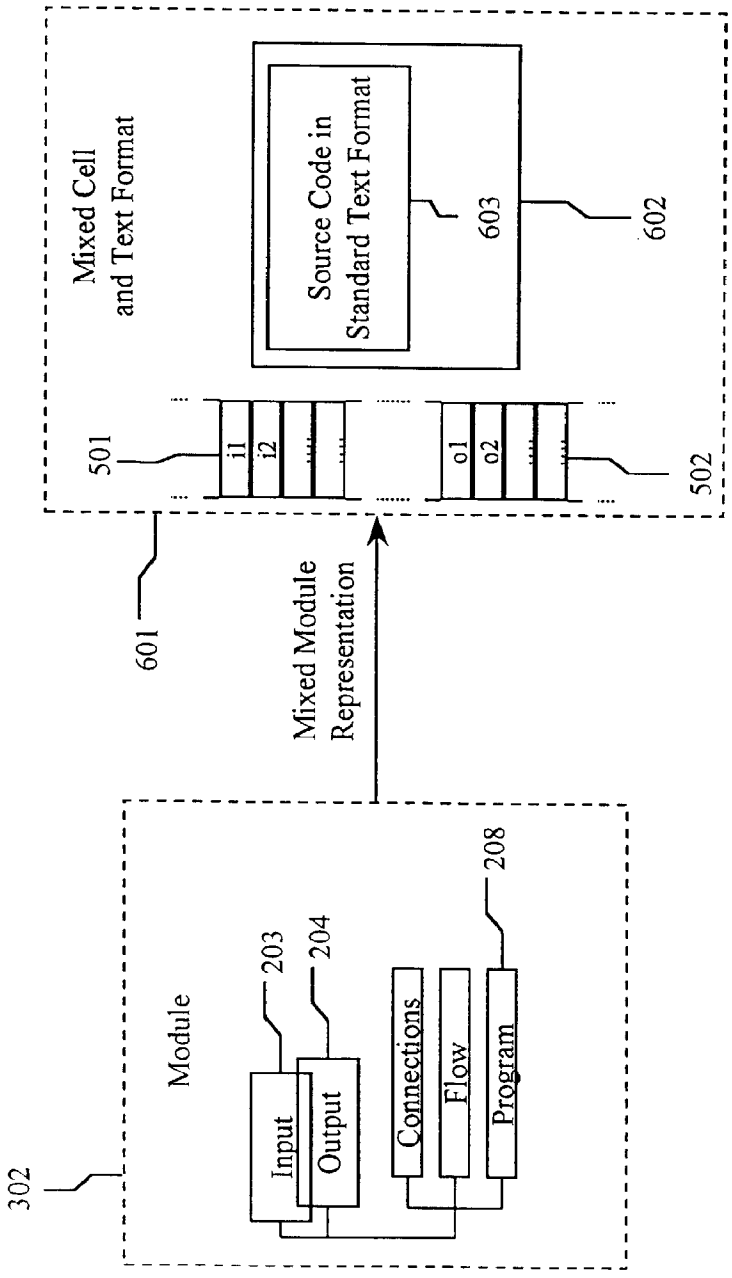
Mixed Cell
and Text Format
Source Code in
Standard Text Format
603
602
501
i1
i2
o1
o2
502
601
Mixed Module
Representation
Module
203
204
208
Input
Output
Connections
Flow
Program
302

Fig. 6

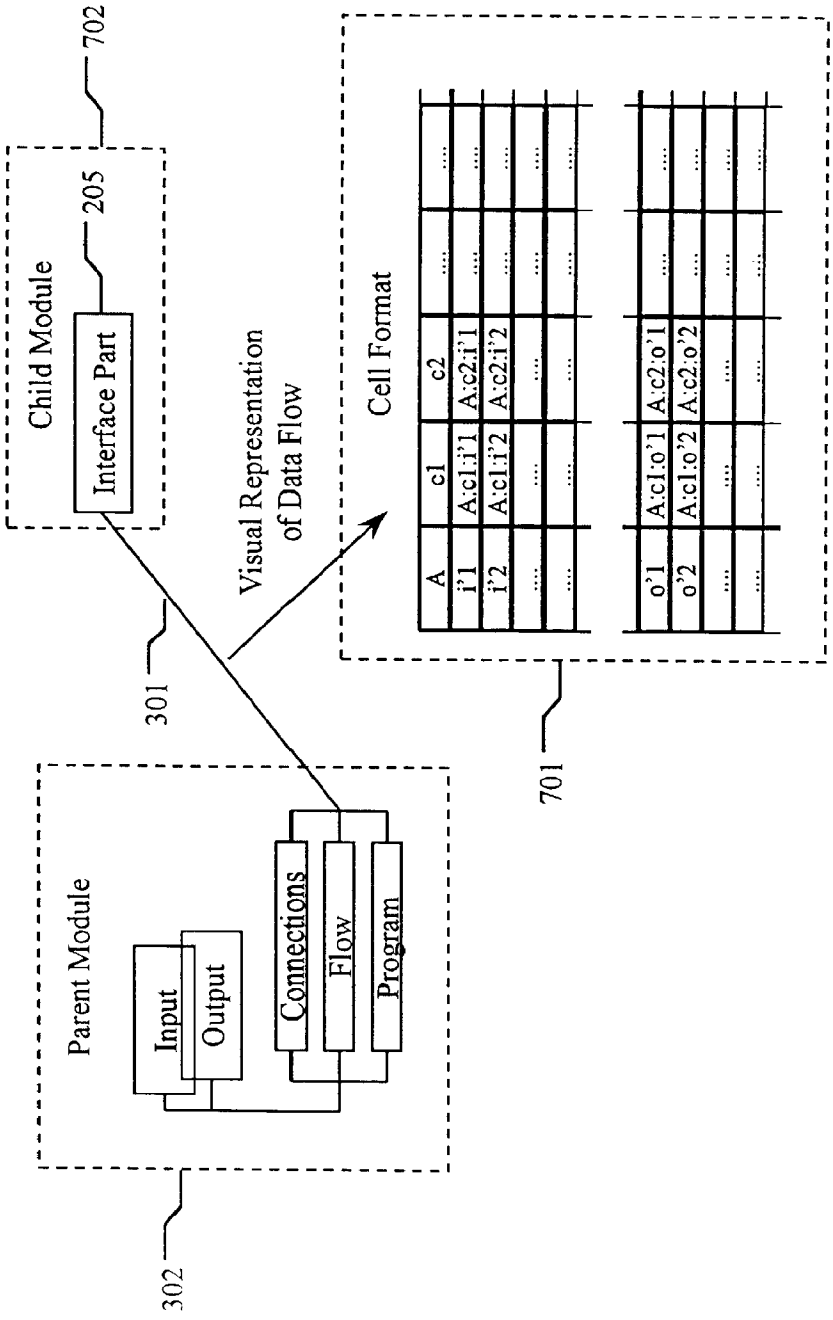
Child Module
Interface Part
205
702
Visual Representation
of Data Flow
Cell Format
A
c1
c2
i'1
A:c1:i'1
A:c2:i'1
i'2
A:c1:i'2
A:c2:i'2
o'1
A:c1:o'1
A:c2:o'1
o'2
A:c1:o'2
A:c2:o'2
301
701
Parent Module
Input
Output
Connections
Flow
Program
302

Fig. 7

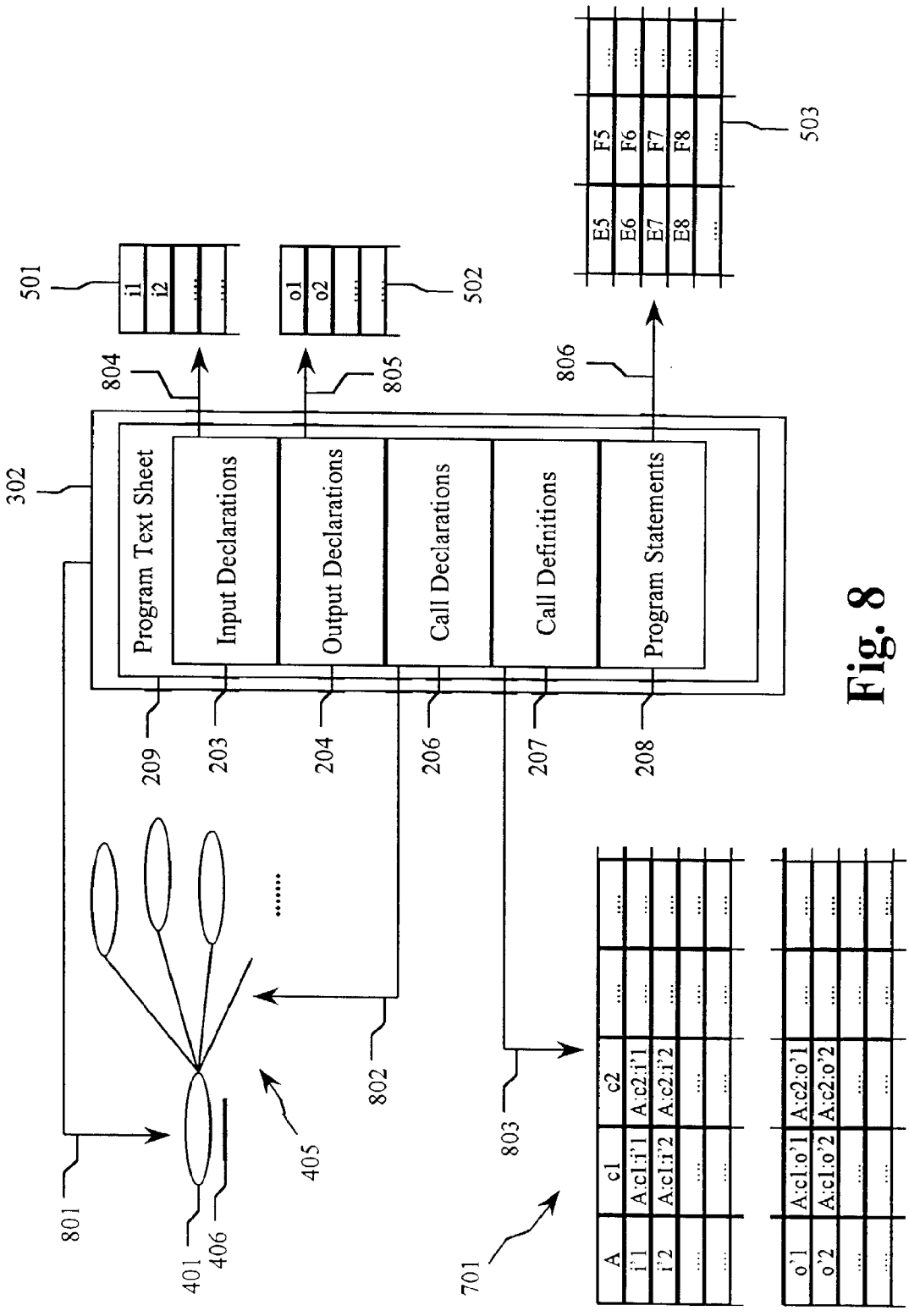
501
i1
i2
502
o1
o2
503
E5
F5
E6
F6
E7
F7
E8
F8
804
805
806
302
Program Text Sheet
Input Declarations
Output Declarations
Call Declarations
Call Definitions
Program Statements
209
203
204
206
207
208
801
401
406
405
802
803
701
A
c1
c2
i'1
A:c1:i'1
A:c2:i'1
i'2
A:c1:i'2
A:c2:i'2
o'1
A:c1:o'1
A:c2:o'1
o'2
A:c1:o'2
A:c2:o'2

Fig. 8

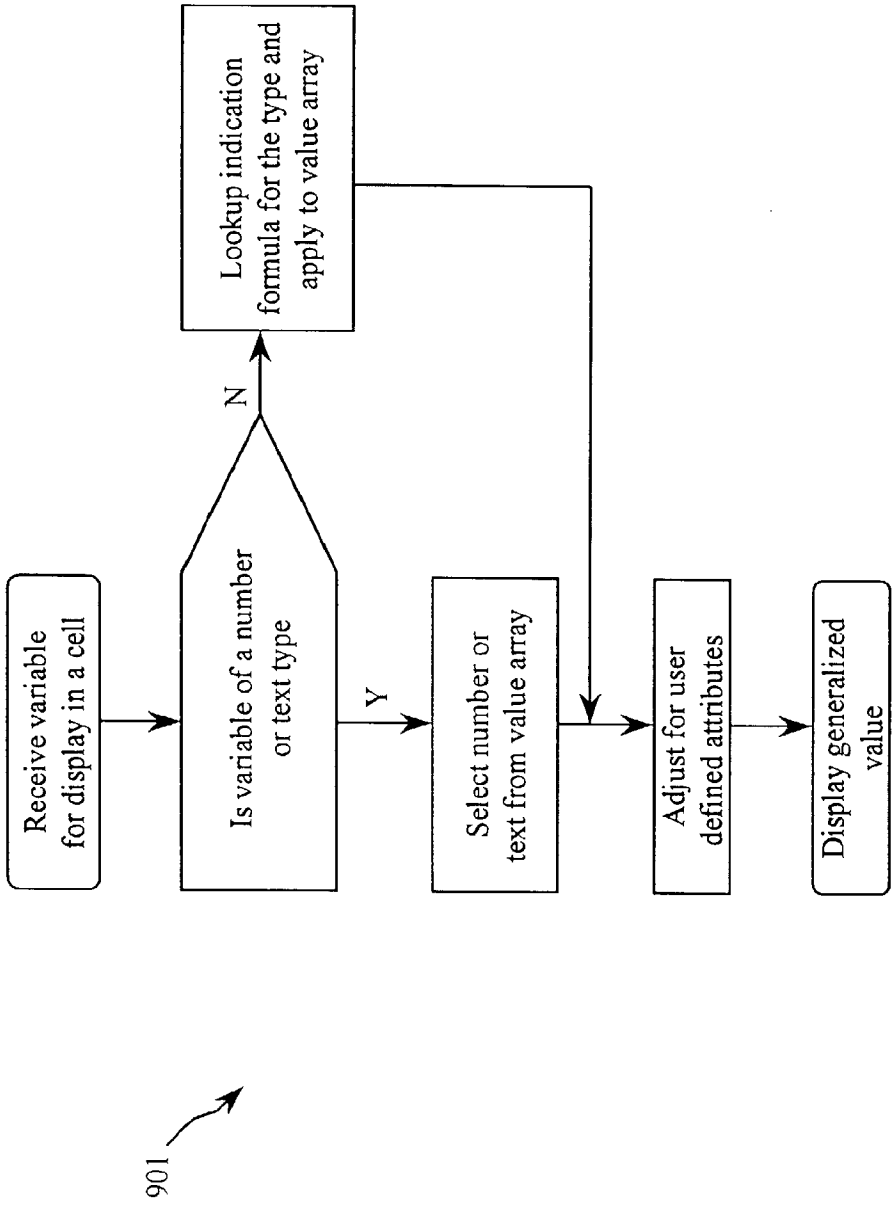
Receive variable for display in a cell
Is variable of a number or text type
N
Lookup indication formula for the type and apply to value array
Y
Select number or text from value array
Adjust for user defined attributes
Display generalized value
901

Fig. 9

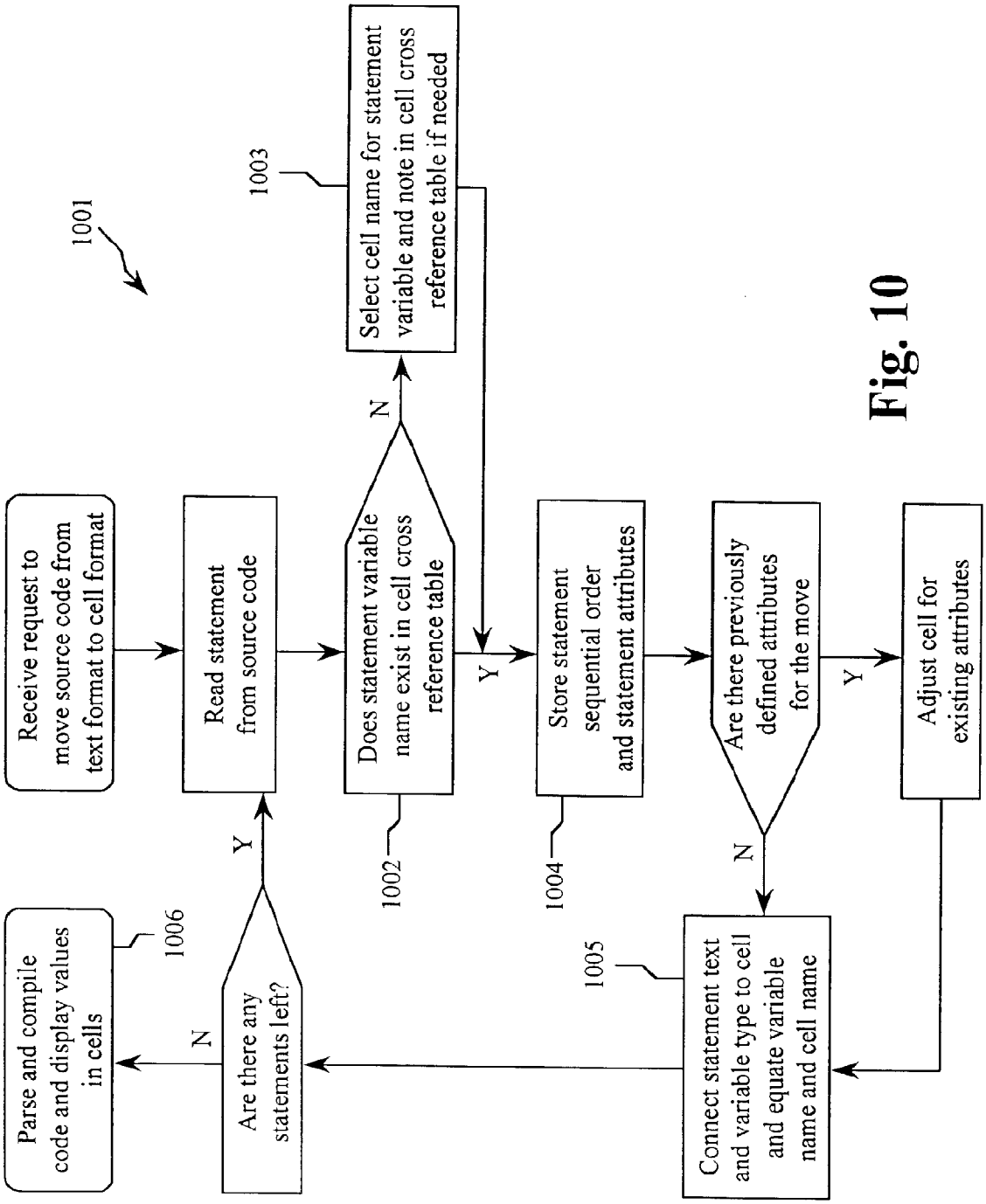
1001
Receive request to move source code from text format to cell format
Read statement from source code
Does statement variable name exist in cell cross reference table
1002
N
Y
1003
Select cell name for statement variable and note in cell cross reference table if needed
Store statement sequential order and statement attributes
1004
Are there previously defined attributes for the move
N
Y
Adjust cell for existing attributes
Connect statement text and variable type to cell and equate variable name and cell name
1005
Are there any statements left?
Y
N
1006
Parse and compile code and display values in cells

Fig. 10

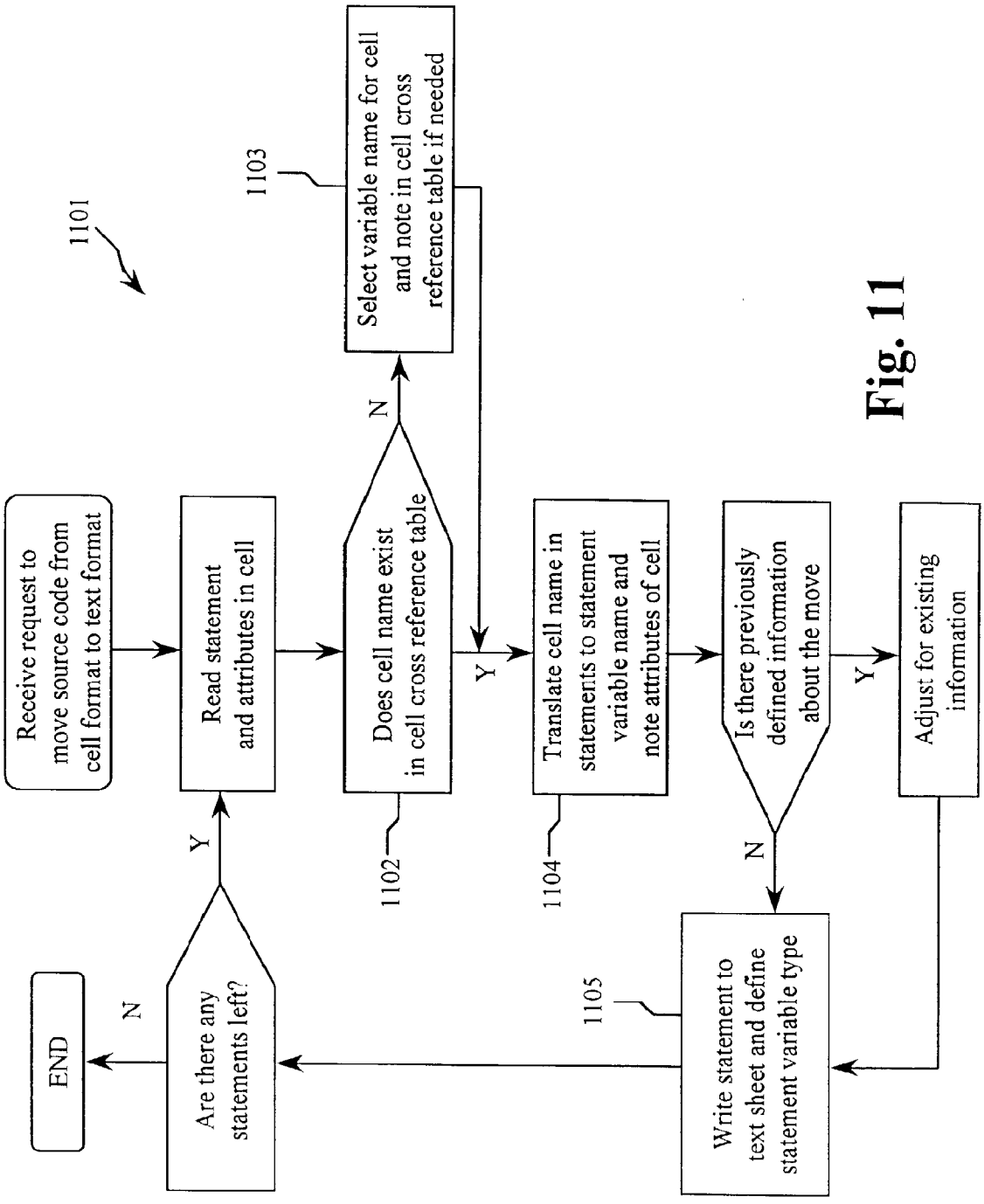
1101
Receive request to move source code from cell format to text format
Read statement and attributes in cell
Does cell name exist in cell cross reference table
N
Y
1102
1103
Select variable name for cell and note in cell cross reference table if needed
Translate cell name in statements to statement variable name and note attributes of cell
1104
Is there previously defined information about the move
N
Y
Adjust for existing information
Write statement to text sheet and define statement variable type
1105
Are there any statements left?
Y
N
END

Fig. 11

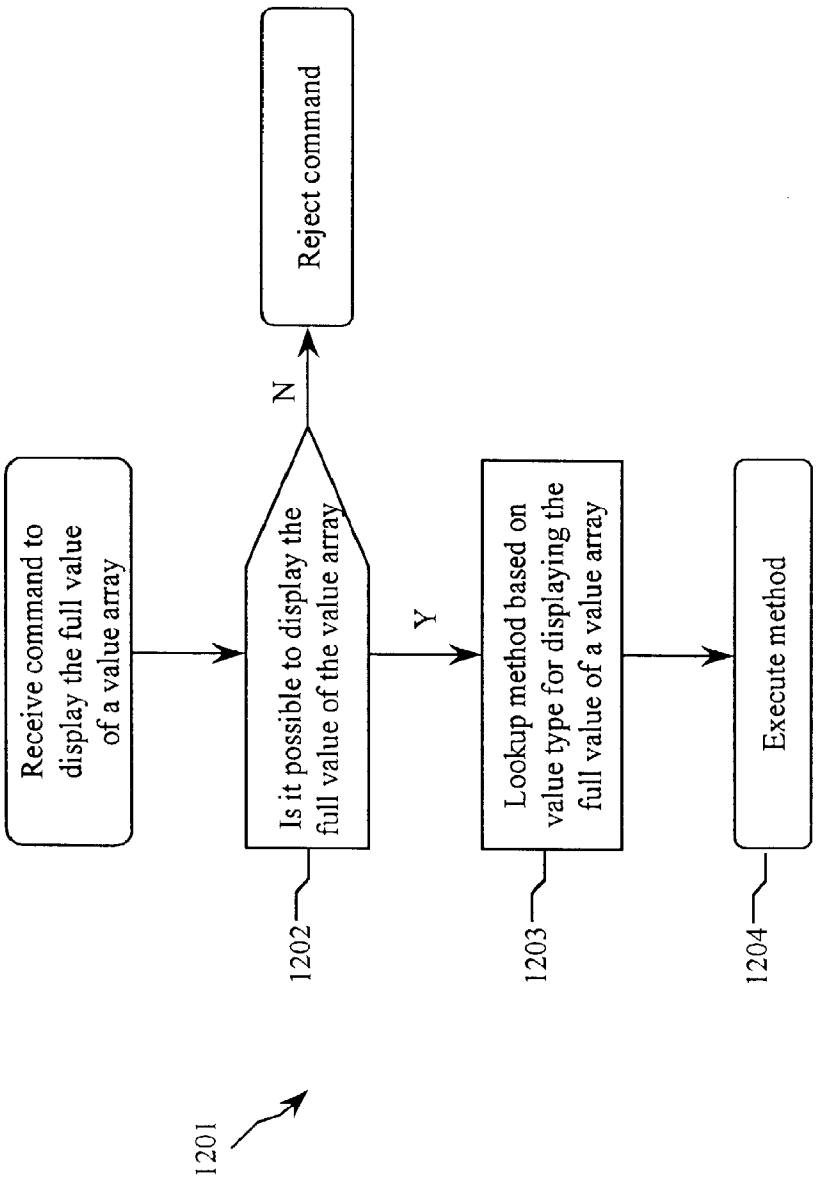
Receive command to display the full value of a value array
Is it possible to display the full value of the value array
N
Reject command
Y
Lookup method based on value type for displaying the full value of a value array
Execute method
1201
1202
1203
1204

Fig. 12

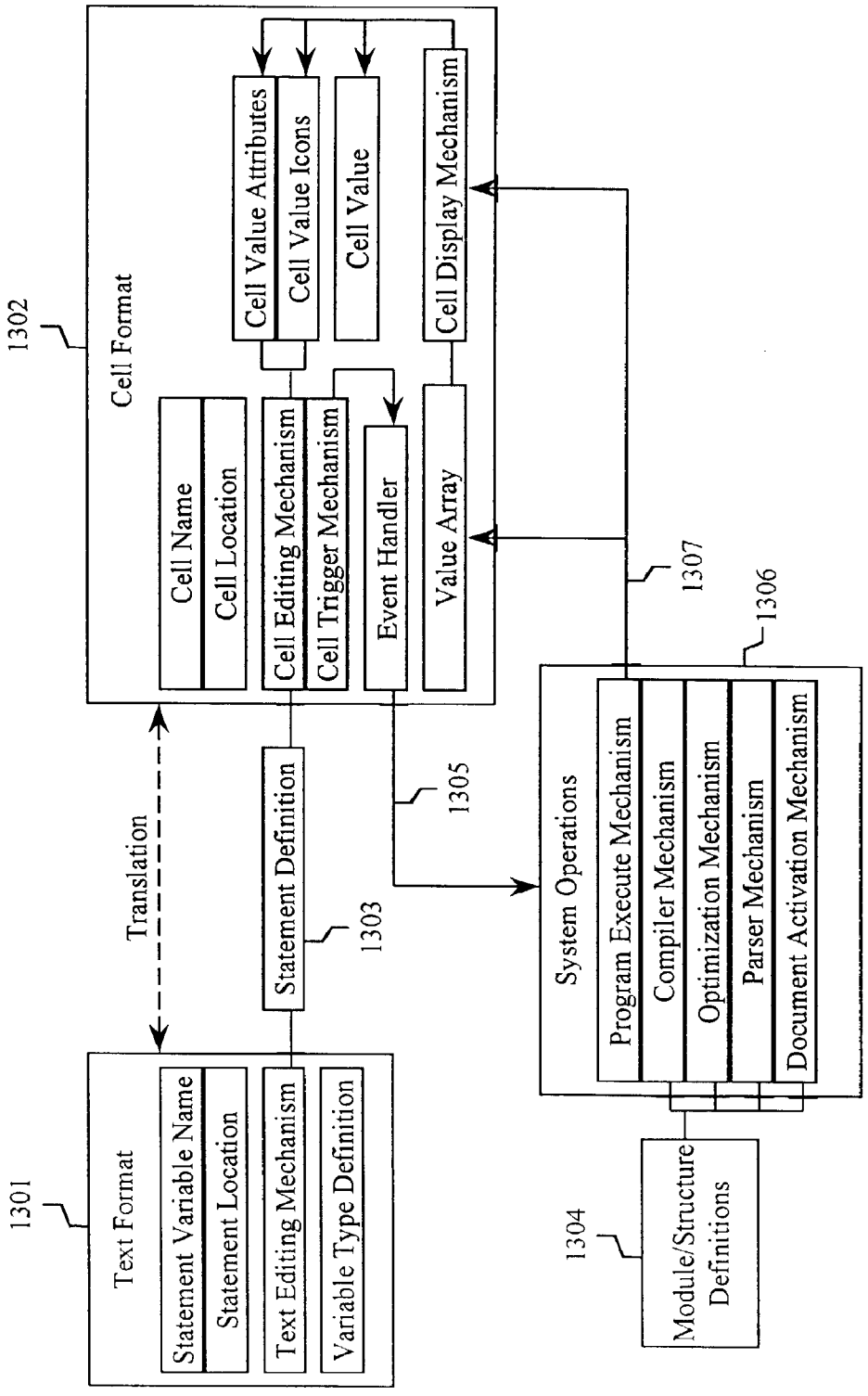
1302
Cell Format
Cell Name
Cell Location
Cell Editing Mechanism
Cell Trigger Mechanism
Event Handler
Value Array
Cell Value Attributes
Cell Value Icons
Cell Value
Cell Display Mechanism
Translation
1301
Text Format
Statement Variable Name
Statement Location
Text Editing Mechanism
Variable Type Definition
Statement Definition
1303
1305
System Operations
Program Execute Mechanism
Compiler Mechanism
Optimization Mechanism
Parser Mechanism
Document Activation Mechanism
1306
1307
1304
Module/Structure Definitions

Fig. 13

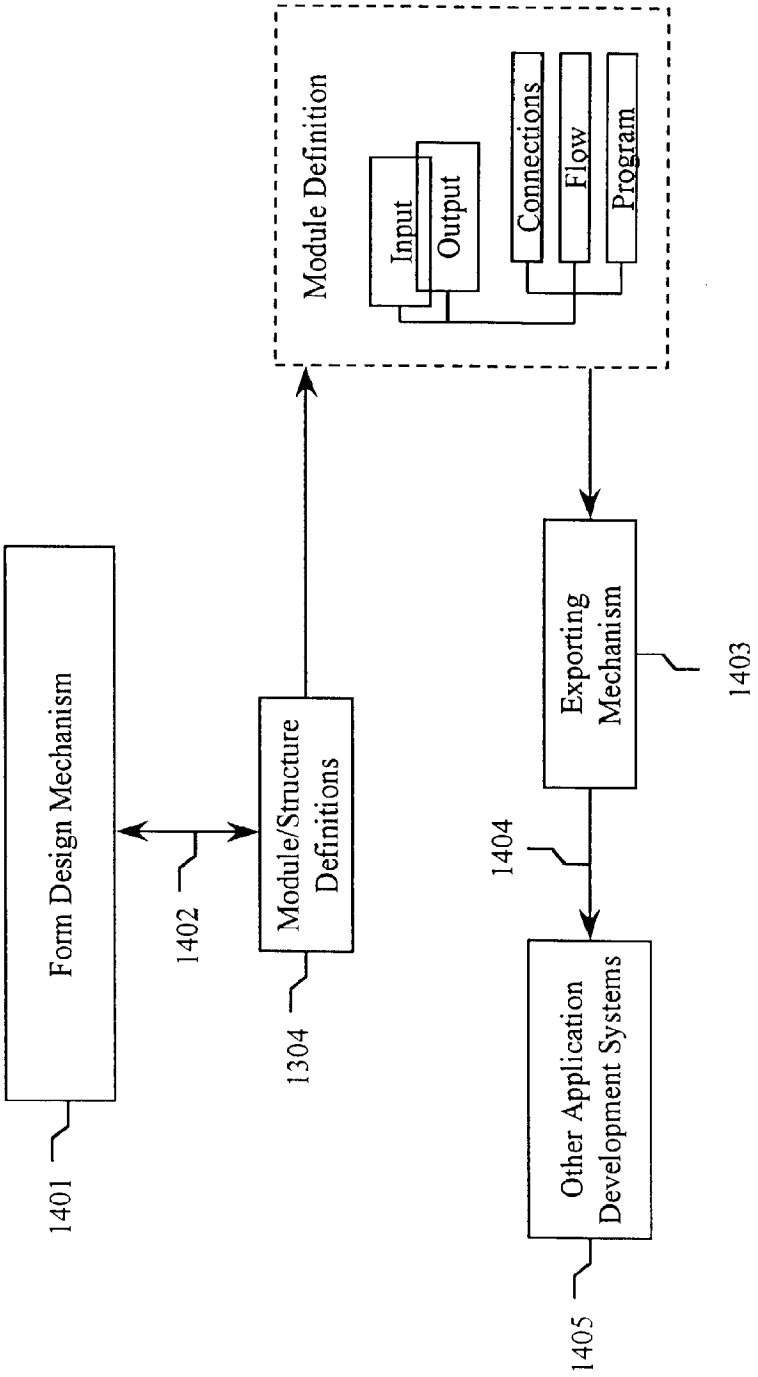
Module Definition
Input
Output
Connections
Flow
Program
Form Design Mechanism
Module/Structure Definitions
Exporting Mechanism
Other Application Development Systems
1401
1402
1304
1403
1404
1405

Fig. 14

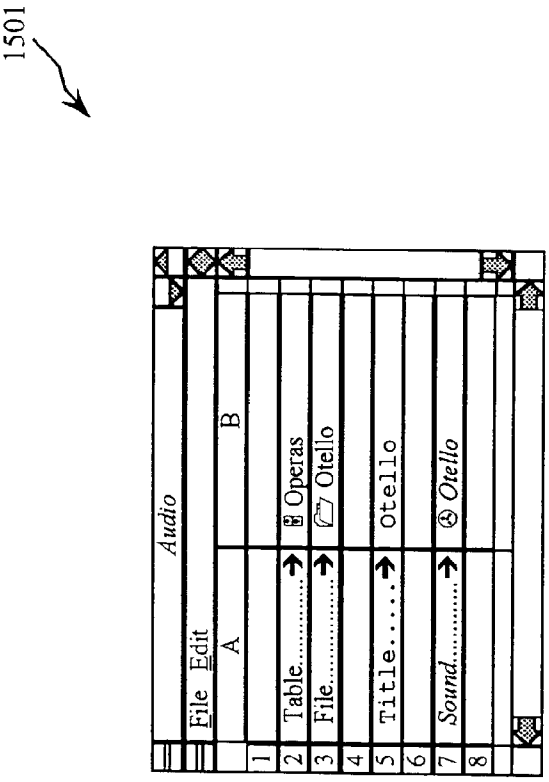
1501
Audio
File Edit
A
B
1
2 Table..... Operas
3 File..... Otello
4
5 Title..... Otello
6
7 Sound..... Otello
8

Fig. 15

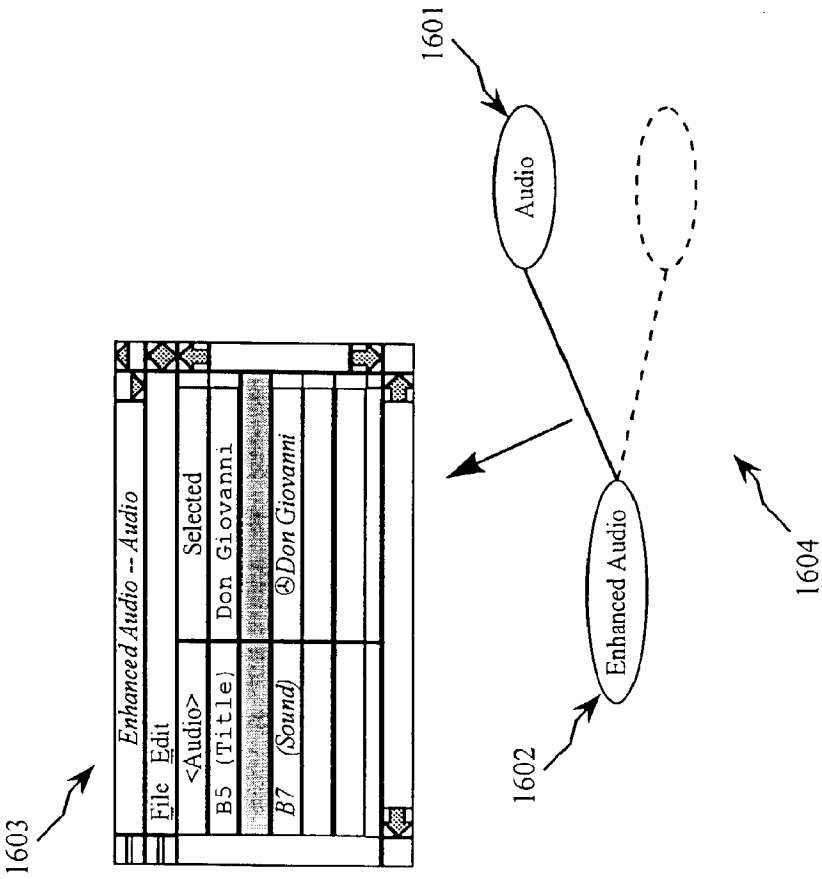
1601
Audio
Enhanced Audio
1602
1603
Enhanced Audio -- Audio
File Edit
<Audio>
Selected
B5 (Title)
Don Giovanni
B7 (Sound)
Don Giovanni
1604

Fig. 16

Research Paper ■

Exploring Performance Issues for a Clinical Database Organized Using an Entity-Attribute-Value Representation

ROLAND S. CHEN, MD, PRAKASH NADKARNI, MD, LUIS MARENCO, MD, FORREST LEVIN, MS, JOSEPH ERDOS, MD, PHD, PERRY L. MILLER, MD, PHD

Abstract **Background:** The entity-attribute-value representation with classes and relationships (EAV/CR) provides a flexible and simple database schema to store heterogeneous biomedical data. In certain circumstances, however, the EAV/CR model is known to retrieve data less efficiently than conventionally based database schemas.

Objective: To perform a pilot study that systematically quantifies performance differences for database queries directed at real-world microbiology data modeled with EAV/CR and conventional representations, and to explore the relative merits of different EAV/CR query implementation strategies.

Methods: Clinical microbiology data obtained over a ten-year period were stored using both database models. Query execution times were compared for four clinically oriented attribute-centered and entity-centered queries operating under varying conditions of database size and system memory. The performance characteristics of three different EAV/CR query strategies were also examined.

Results: Performance was similar for entity-centered queries in the two database models. Performance in the EAV/CR model was approximately three to five times less efficient than its conventional counterpart for attribute-centered queries. The differences in query efficiency became slightly greater as database size increased, although they were reduced with the addition of system memory. The authors found that EAV/CR queries formulated using multiple, simple SQL statements executed in batch were more efficient than single, large SQL statements.

Conclusion: This paper describes a pilot project to explore issues in and compare query performance for EAV/CR and conventional database representations. Although attribute-centered queries were less efficient in the EAV/CR model, these inefficiencies may be addressable, at least in part, by the use of more powerful hardware or more memory, or both.

■ **J Am Med Inform Assoc.** 2000;7:475–487.

Affiliations of the authors: Yale University, New Haven, Connecticut (RSC, PN, LM, PLM); Evergreen Design, Guilford, Connecticut (FL); and Veterans Affairs Medical Center, West Haven, Connecticut (JE).

This work was supported in part by NIH grants T15-LM07056 and G08-LM05583 from the National Library of Medicine and by grant U01-CA78266 from the National Cancer Institute.

Correspondence and reprints: Prakash M. Nadkarni, MD, Center for Medical Informatics, Yale University School of Medicine, P.O. Box 208009, New Haven, CT 06520-8009; e-mail: ⟨prakash.nadkarni@yale.edu⟩.

Received for publication: 10/26/99; accepted for publication: 03/06/00.

A problem that data modelers commonly encounter in the biomedical domain is organizing and storing highly diverse and heterogeneous data. For example, a single patient may have thousands of applicable descriptive parameters, all of which need to be easily accessible in an electronic patient record system. These requirements pose significant modeling and implementation challenges.

One increasingly popular solution that addresses these issues is to model data using the entity-attribute-value (EAV) approach. This model, which was historically seen first in LISP association lists,[1] has

been used in multiple applications, including the HELP system[2,3] and the Columbia-Presbyterian Clinical Data Repository.[4,5] The EAV approach offers many advantages, including its flexibility and ability to store heterogeneous data in a simple, easily maintained format. Our group has recently enhanced this representation to permit data modeling with classes and relationships (EAV/CR).[6]

Despite its significant benefits, EAV design has the potential to be less efficient than "conventional" database schemas when accessing data. In particular, attribute-centered queries, where the query criterion is based on the value of a particular attribute, are most likely to show impaired performance. This is especially true when query criteria combine one or more simple conditions in Boolean fashion. An example of such a query is "find all patients with blood cultures positive for either Streptococcus pneumoniae OR Candida albicans." The reason for the potential performance degradation is that the relatively fast AND, OR, and NOT operations that would be required if we were operating on conventional schema tables must be converted to considerably slower set-based equivalents (set intersection, union, and difference, respectively) for EAV tables.

Attribute-centered queries are important for research questions; their performance is not critical for the care of individual patients. The contents of the production (patient-care) database serve, however, as the basis for any research databases that an institution must support, and the question is how such research databases must be implemented. There are two ways to do so:

- The simplest solution is to periodically take a backup copy of the production database and restore it onto separate hardware, with only modest transformation, e.g., addition of indexes. (It is undesirable to run resource-intensive attribute-centric queries directly on the production system because they would take CPU cycles away from the simple but critical queries that assist management of individual patients.)
- An alternative solution is to redesign the schema completely as numerous conventional tables on separate hardware and transform the production EAV data prior to loading it into these tables. The tables may reside in a single research database or in multiple, special-purpose databases. One could extract all the data or only a subset, focusing on the attributes of greatest clinical or epidemiologic interest. Such a procedure is followed at Intermountain Health Care, where multiple conventionally structured data marts are populated by export of data subsets from the HELP repository,[17] whose contents are mostly in EAV form.

The first solution is easy to set up, but queries may not perform well. The second solution promises better performance but is more elaborate because of the data transformations involved. The required transformations can be extensive if one has to support multiple research efforts. To determine which solution is applicable in given circumstances, it would be useful to have performance metrics that compare the relative performance of attribute-centric queries on EAV schemas and equivalent queries on conventional schemas.

We have not found any published reports that quantify these performance differences systematically. Of related interest are the influences on query performance exerted by factors including database size and hardware configuration (e.g., amount of available physical memory). The questions addressed in this paper are:

- Does degradation of query performance occur and, if so, how severely?
- Can degradation, if present, be mitigated by database or query design strategies and, if so, to what extent?

Answers to these questions would provide valuable insights into the viability of the EAV approach for supporting large-scale databases.

In this paper, we explore issues in database performance for real-world microbiology data stored using an EAV/CR and conventional approach. In particular, we examine: 1) the differences in execution time for a set of clinically relevant database queries operating on an EAV/CR and conventional schema containing the same set of data, 2) the relative efficiency of different EAV/CR query strategies, and 3) the effect of database size and physical memory on database performance. Finally, we examine monitors of system performance and identify potential performance bottlenecks for the competing database schemas.

Background

The EAV model, which has been described in great detail elsewhere,[7–9] can be visualized conceptually as a database table with three columns: 1) "Entity ID" or "Object ID"; 2) "Attribute," or a pointer to a separate "Attributes" table; and 3) "Value," containing the value corresponding to the particular entity and attribute. In an electronic patient record system, the entity ID usually refers to a specific patient-associated

event stored as a patient ID and time stamp. The attribute and value columns correspond to a specific parameter (e.g., potassium) and its value, respectively. Each entity-attribute-value triplet occupies one row in the table. Since attributes are not represented as columns, this single table can store all values in the database.

EAV/CR enhances the basic EAV approach by using the object-oriented concepts of "classes" and "relationships" to facilitate data modeling. Classes in EAV/CR are data structures that store the attributes (possibly including other classes) associated with a particular type of entity. For example, a bacterial isolate from a blood culture could be modeled with the class "Bacterial Organism" that contains attributes "Organism_ID" (which references an organism name), "Quantity_cultured", and "Antimicrobial_tested." (The last attribute is multivalued, because multiple antimicrobials are typically tested against a single culture.) "Antimicrobial_tested," in turn, is itself a class with the attributes "Antimicrobial_ID" (which references a drug name) and "Sensitivity_result."

As in object-oriented modeling, objects represent instances of a class. Thus, in our previous example, every cultured bacterial isolate possesses an object ID as an instantiation of class "Bacterial Organism." Relationships are used to describe inter-class interactions. Metadata, or information about the data contained in the database, is then applied to describe, maintain, update, and access the data. This overall approach allows the system to incorporate new attributes without altering the underlying physical database schema.

Conventional schemas, in contrast, model parameters as distinct columns. For example, the parameter "Organism_name" could be represented as a column in a table "Microbial_Organism." As the database evolves to accommodate diverse data, the number of parameters grows and the number of columns and tables that are needed increases. Thus, with a rapidly evolving and expanding domain such as medicine, where new tests and concepts are commonplace, the underlying schema requires frequent modification. This is particularly true of clinical medicine, where there can be dozens of specialty-specific tables, each with a constantly growing list of fields.

The ability of the EAV approach to store diverse data in a single (or few) tables greatly simplifies the physical schema of the database. It provides an efficient way to access all data pertinent to a specific object (i.e., entity-centered queries). In an EMR, this often takes the form of "Select all data (clinical findings, lab tests, etc.) for patient X." In a conventional schema with numerous tables, the user would need to search every table containing patient data to extract the relevant data for the patient, making query development laborious. Furthermore, not all these tables may contain data for a given patient; as a result, many tables may be searched unnecessarily.

As stated previously, it is the attribute-centered queries that are of concern to the developer who is focused on efficiency. To systematically quantify the performance of an EAV/CR schema compared with its conventional counterpart, we chose the domain of microbiology. This domain, which has been the subject of many previous database models,[10,11] provides the potential to work with complex data structures where class attributes exist as both primitive types (e.g., strings) as well as other classes.

Methods

In this section, we describe our test database and schema and various aspects of the evaluation strategy that we devised. In each case we provide the rationale of our design decisions.

Data Description

Each specimen collected from a patient in the Veterans Administration (VA) system is assigned a unique ID. The ID of the ordering physician is also recorded against the specimen. One or more tests (e.g., Gram stain and culture) may then be run on each specimen. For each culture, a panel of antimicrobial sensitivity tests is performed. The data are stored in DHCP,[12] an M language database that serves as the clinical patient repository used nationwide by VA Medical Center (VAMC) hospitals.

The entity-relationship diagram for the original data is shown in Figure 1 (details have been omitted for clarity). All classes/entities can be traced to the class "Microbiology Specimen," which sits at the "top" of the class diagram. Classes contain either primitive types or other classes as attributes, which enable classes to relate to one another.

It should be noted that the entities "bacterial culture," "fungal culture," etc. shown in Figure 1 are represented in separate tables in DHCP. During the process of data transformation, we merged these into a single table, since the separate tables have similar structures.

The original, raw, antimicrobial sensitivity data had the names of antimicrobials (e.g., "streptomycin," "chloramphenicol") hard-coded as columns in a table. This represents poor table design. With each new antimicrobial introduced into the institution, the table

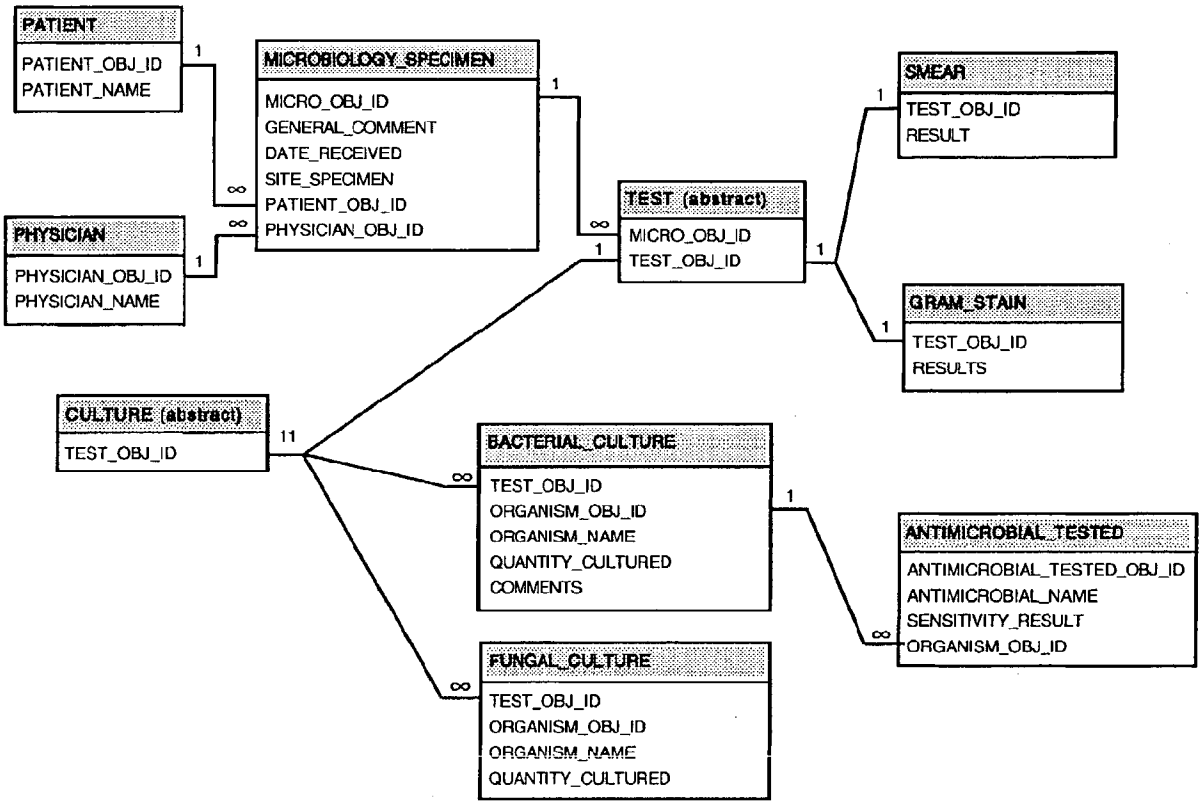

Figure 1 Entity-relationship diagram for the microbiology data. For reasons of space, some tables, which are not relevant to the queries in the text, are not shown.

structure needs updating, and all forms dependent on the table require redesign with each change. It may be possible to use this design in the M language, where all disk-based data structures are sparse by design. In relational database management systems (RDBMSs), however, such a design wastes much space, because NULL values of empty columns are still recorded. Further, RDBMSs have a limit on the maximum number of columns per table (e.g., 255), which might be exceeded in some circumstances. Even if space and column constraints are unimportant, necessitating alterations in the table structure and application code each time the number of objects changes is not a good system design approach.

Other parts of the DHCP schema are correctly designed. For example, in the Pharmacy subschema, names of medications are not hard-coded as columns but treated as data: the ID of the drug in the Orders table references a list of drugs in a Drugs table. Therefore, to make comparisons between EAV and conventional schemas more realistic, we first transformed the sensitivity data into the (correct) row-modeled form, where the antimicrobial ID was an attribute in a column and referenced a table of antimicrobial drug names.

Data Extraction

We extracted a dataset from VA Connecticut DHCP that contains all the available online data, which range from 5/1/87 to 9/26/98 and include results from more than 135,000 microbiology test specimens and more than 400,000 antimicrobial sensitivity tests for more than 28,000 patients. The extracted data were transformed for storage into a SQL Server 7.0 RDBMS. The extraction was done in two steps. We first created a conventional schema from the data (transforming the data where appropriate, as described above). We then created an EAV/CR schema from the conventional one by assigning a unique object ID to each instance of a particular class. In all, there were 965,529 instantiated objects in our database.

The Conventional Schema

The conventional schema is shown in Figure 2. All primary and foreign keys, as well as fields used for

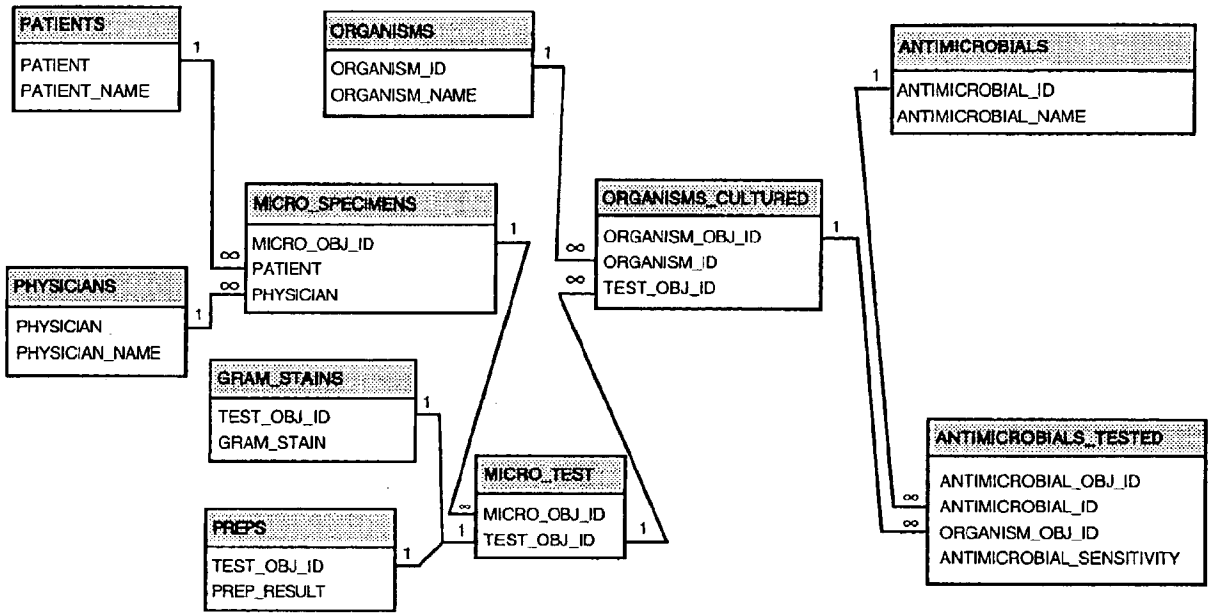

Figure 2 The conventional physical database schema. Fields in some tables are not shown, for reasons of space; for the most part, only fields linking to other fields are illustrated.

search (e.g., antimicrobial and organism names), are indexed.

It should be noted that the culture and antimicrobial sensitivity tables can be considered special-purpose or homogeneous EAV tables (because each stores only a single class of data in row-modeled form). In contrast, "EAV" databases contain general-purpose tables that store heterogeneous data (across many classes).

The EAV/CR Schema

The EAV/CR physical schema is shown in Figure 3. As in all EAV/CR schemas, there are two parts to the schema: metadata and data. (The EAV/CR metadata schema are not illustrated. They have been described in detail,[6] and a summary is available on the Web at http://ycmi.med.yale.edu/nadkarni/eav_cr_frame.htm). Attributes are segregated by data type (e.g., string values are stored in "EAV_TEXT" whereas dates are held in "EAV_DATE").

The table EAV_OBJECTS is of particular interest. It accommodates situations where one object can contain another object and permits objects to be associated with one another. In this table, the "Value" field is the ID of a "child" object. Thus, in our logical schema, tests are children of a specimen, cultures are children of a test, and antimicrobial sensitivity results are children of a culture. In other words, when data are hierarchic (as in consecutive one-to-many relationships), each record in EAV_OBJECTS represents one node of a tree. To traverse down an object hierarchy starting with a particular object, we locate that object's ID in the Object_ID column of EAV_OBJECTS. We then gather all Values (child Object IDs) for this entity. We then recurse, searching for these IDs in the Object_ID column, and so on.

One well-known method of minimizing the number of recursive queries when traversing an object hierarchy is to redundantly store the ID of the "ancestor" object against every record in a tree, so that all descendants of the ancestor can be retrieved in a single SQL statement.[13] If a significant percentage of queries also use the "ancestor" class, the number of joins required to retrieve the data is also minimized. To allow us to explore this means of schema optimization and its impact on queries, we created an "ancestor" field containing the microbiology specimen ID associated with each test value in EAV_TEXT.

We indexed all object ID columns and created a compound index on attribute-value pairs for tables EAV_TEXT and EAV_DATE, since queries typically select data using both an "attribute ID" and "value" together as search criteria. In addition, a separate index was created for the Value column of EAV_OBJECTS, since the nature of a query might make it desirable to traverse an object hierarchy from descendant to ancestor.

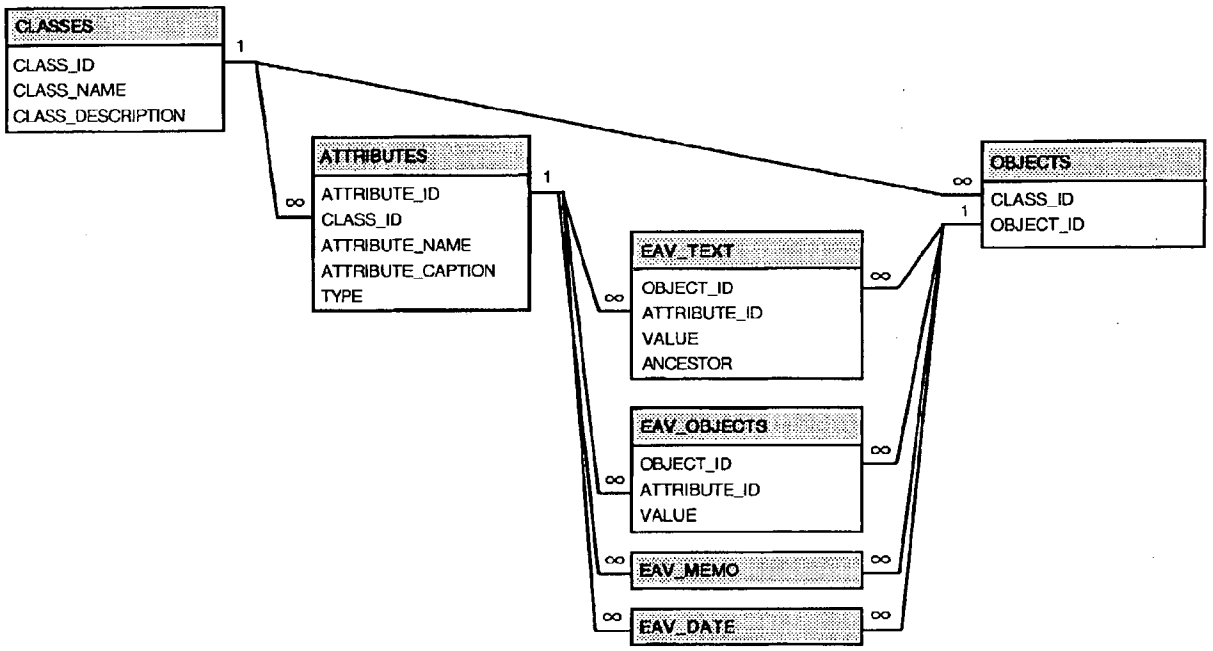

Figure 3 EAV physical database schema. Since all EAV tables share the same structure, the details of two tables have been omitted.

Expanding the Size of the Data

To measure the effects of database size on performance, we replicated the contents of our initial two databases (EAV/CR-Small and Conventional-Small) and reassigned newly created, unique IDs for each new object. The replicated data were then appended onto the corresponding, original database to create "new" databases with twice the number of objects (EAV/CR-Medium and Conventional-Medium). This process was then repeated to yield databases four times larger than our originals (EAV/CR-Big and Conventional-Big). Since the underlying schemas did not change, queries did not require modification within a given representation. The "Big" data set represents the equivalent of 30 to 40 years' worth of microbiology data for all patients in VAMC Connecticut.

The benchmark timings for EAV/CR-Medium and Conventional-Medium were intermediate between the smallest and largest representations, with the data showing an approximately linear trend between the "small" and "big" databases. The purpose of studying three database sizes was to allow us to assess better how relative performance varied with size. To minimize information overload in the tables presented in the remainder of the paper, we omit further discussion of the two "medium" databases.

The EAV/CR representation consumed approximately four times the storage of our conventional schema. EAV/CR-Big consumed 1,177 Mb, with indexes accounting for more than 62 percent. Our largest conventional representation (Conventional-Big) was 301 MB, with approximately 32 percent accounted for by indexes.

It is true that EAV/CR is more space-efficient for sparse data. In this case, however, prior to importing the raw data, we had transformed the bulk of the original sparse data (e.g., the antimicrobial sensitivity results and the cultures) into dense, row-modeled facts in columns that are mostly IDs (long integers). It is well known that a row-modeled conventional table will always take significantly less space than the equivalent facts represented in EAV form. This is because a single set of facts is represented as a single row in a row-modeled table but as multiple rows in an EAV table, one per fact. For example, to fully describe a single antimicrobial sensitivity result, we store several related facts linked to the specimen ID: the micro-organism that was isolated, the antimicrobial tested, and the sensitivity of the former to the latter. In the EAV representation, each row has the extra overhead of Object ID and Attribute ID, plus accompanying indexes.

Query Benchmarking

To compare performance between the two competing physical representations, we developed several clini-

cally oriented queries, as follows. The two attribute-centered queries were:

- Query 1: Find all patients with cultures positive for *Pseudomonas aeruginosa.*
- Query 2: Find all specimens that grew *Streptococcus pneumoniae* and *Candida albicans.*

The EAV/CR approach, as employed in electronic patient record systems, is generally known to be efficient for patient-centered queries (which correspond to queries based on the Object ID rather than on the Attribute and Value). However, we wanted to verify that this was indeed the case for our data set and determine the performance penalty, if any, in comparison with the conventional schema. Therefore, we also created two entity-centered queries:

- Query 3: Find all microbiology tests run for a specific patient.
- Query 4: Find all antimicrobials and sensitivities tested for a particular culture.

System Description: Memory Allocation

Tests were conducted on a Dell Dimension XPS D300 running Windows NT Server 4 with a 300-MHz Pentium II processor and 66-MHz system bus. Execution times were recorded using the SQL Server 7.0 Profiler tool.

We ran our benchmarks with 192 Mb and 256 Mb of physical RAM and allocated 64 Mb and 128 Mb, respectively, to SQL Server 7. We did this to ensure that the additional RAM was specifically allocated to the database. This left 128 Mb of RAM for the operating system and its associated tasks. The system employed a Western Digital UDMA, 9.5mS, 5400 RPM hard drive. We want to emphasize, however, that in our benchmarks it is the ratio of timings between the EAV/CR and conventional schemas that are important, rather than the absolute timings.

Alternative Formulation of EAV/CR Queries

Formulation of a complex query against an EAV/CR database is significantly more difficult and error-prone than is formulation of a functionally identical SQL query against an equivalent conventional schema. This is because the physical schema of an EAV/CR database—the actual tables holding the data—differs markedly from its logical schema, which specifies how the classes modeled in the database are related to each other. For example, when considering Query 1, the "culture," "organism," and "patients" tables do not physically exist in the EAV/CR schema. In a conventional schema, in contrast, logical and physical schemas are identical.

Queries against an EAV/CR system must, therefore, be formulated in terms of the semantics of the logical schema and must then be translated into operations on the tables in the physical schema. We present four ways to do this, the third and fourth being slight modifications of the second, as follows:

Method One

A single (large) SQL statement can be created, which extracts the desired data by joining the necessary tables. In an EAV/CR schema, however, a small set of physical tables is used to model many more logical tables. This results in the same tables being used repeatedly under different aliases, with "self-joins" being performed. In Figure 4, which illustrates this approach for Query 1, the table EAV_TEXT is used twice. As the logical schema becomes more complex and more classes have to be traversed, such SQL gets progressively harder to write and debug, because it often incorporates a large number of nested constructs.

To complicate matters, when this SQL is sent to an RDBMS, the latter parses the code and tries to formulate an optimal strategy before executing it. Query optimization is a problem known to be factorial with respect to the number of tables or aliases participating in the join.[14] (Factorial 8, or 8!, equals $8 \times 7 \times 6 \ldots \times 2 \times 1 = 40{,}320$. An excellent discussion of the query optimization problem is posted at http://www.informix.com/informix/solutions/dw/redbrick/wpapers/star.html.)

Difficulties manifest, therefore, as the number of aliases increase. While DBMS optimizers are claimed by their vendors to use clever heuristics in such a situation, simply evaluating each of the numerous alternatives takes time. (In an experiment reported by Nadkarni and Brandt,[9] an attempted 20-alias join that would have yielded only 50 rows in the result set caused an RDBMS to freeze.)

The only justifiable circumstances for using a single-statement approach for EAV/CR query are for queries with a modest number of aliases (as in Figure 4), or for "stored" queries, a facility available in many DBMSs. (In execution of stored queries, the parsing phase is bypassed, because the query has already been "compiled," so to speak, with a plan of execution already worked out.)

Method Two

The second method uses a "divide and conquer" strategy. A series of simple queries are specified in

```
select distinct EAV_TEXT1.VALUE
from EAV_TEXT inner join
  EAV_OBJECTS on
  EAV_TEXT.ANCESTOR = EAV_OBJECTS.OBJECT_ID inner join
  EAV_TEXT as EAV_TEXT1 on
  EAV_OBJECTS.VALUE = EAV_TEXT1.OBJECT_ID
where (EAV_TEXT.VALUE = 'PSEUDOMONAS AERUGINOSA')
  and (EAV_TEXT.ATTRIBUTE_ID = 20)
  and (EAV_TEXT1.ATTRIBUTE_ID = 9)
```

Figure 4 Database query developed using a single SQL statement. In this and the next three figures, attribute ID 20 refers to organism name and ID 9 refers to patient ID.

terms of individual classes in the conceptual schema. Each query accesses one or two tables at a time to create a temporary table. The temporary tables are then combined with joins, using appropriate set operations (intersection, union, difference), if necessary. When the final result is obtained, temporary tables created along the way are deleted.

In this particular case, the DBMS optimizer has no problem with individual queries. In fact, the execution plan devised by optimizers often involves creating temporary tables, and we are explicitly telling the DBMS what to do at each step rather than relying on the DBMS to determine its own strategy. Figure 5 illustrates this method for Query 1.

It is, of course, possible that, if a DBMS spends enough time analyzing a complex query, it will arrive at a solution superior to what we have manually specified. However, for highly complex ad hoc queries (unlike stored procedures), the time required to devise an execution plan may be much greater than the time required to actually execute it.

Method Three

A slight modification of the second method is to repopulate and empty existing indexed temporary tables instead of creating and deleting them. The potential rationale is that if intermediate result sets created by a simple query are large, then, when the result sets are to be combined, the query optimizer will often create temporary indexes to speed up the join process. If we maintain several indexed tables for the sole purpose of holding intermediate results, then, when the tables are populated through INSERT queries, the indexes are also populated, and subsequent joins will be speeded up. Figure 6 illustrates this approach. Notice the very great similarity to the code in Figure 5. The only differences are that we use INSERT INTO ... SELECT instead of SELECT ... INTO, and TRUNCATE TABLE instead of DROP TABLE.

This strategy, however, is double-edged. If intermediate results are small, then the DBMS may perform joins in memory rather than on disk. If so, we have created disk-based indexes for nothing. Index maintenance uses machine resources and time, because whenever rows are removed or added to a table, the indexes on the table must also be maintained.

Method Four

A minor modification of the third method is to use initially un-indexed temporary tables, which are indexed only after all records are inserted. Record insertion is expected to be faster because the indexes do not have to be maintained each time a new record is added to the corresponding table. The query is illustrated in Figure 7.

We applied all four methods to Query 1 operating on EAV/CR-Big with 128Mb RAM allocated to the database. We found that our second method (developing simple queries that created tables and later removed them) gave the best results.

Measuring Caching Effects

When the exact same query is sent to a DBMS multiple times, the execution times for the second and subsequent runs are often significantly shorter than for the first. This is because the DBMS uses caching, either in memory or on disk. The SQL string of a query is stored (so that the text of a new query may be compared with it), along with the execution plan.

```
select EAV_TEXT.ANCESTOR into TEMP1
  from EAV_TEXT
  where EAV_TEXT.VALUE="PSEUDOMONAS AERUGINOSA"
   and EAV_TEXT.ATTRIBUTE_ID=20;
select EAV_OBJECTS.VALUE into TEMP2
  from TEMP1 inner join EAV_OBJECTS on TEMP1.ANCESTOR =
EAV_OBJECTS.OBJECT_ID;
select distinct EAV_TEXT.VALUE
  from TEMP2 inner join EAV_TEXT on TEMP2.VALUE =
EAV_TEXT.OBJECT_ID
  where EAV_TEXT.ATTRIBUTE_ID=9;
drop table TEMP1; drop table TEMP2;
```

Figure 5 Database query developed using temporary tables created and deleted dynamically to store interim data.

If the result set is small enough, it may also be cached. Therefore, in an extreme case, a repeated query will not cause any disk activity at all: the DBMS, having decided that it is identical to a previous one, will instantaneously return a stored result.

It is important to determine caching effects for conventional versus EAV/CR schemas, especially if queries on the latter consist of a batch rather than a single statement. (All the individual queries in a batch might not be cached.) To measure caching effects, each query was executed five times in succession immediately following system start-up for each combination of system memory and database size. We used five runs to control for "cold" database, or start-up effects.[15]

We report the timings for every query result in two ways: as initial query execution time and as cached execution time (the average of the four succeeding runs). The former probably simulates the real-world situation more closely than the latter, since it is less likely that two different users of the system will perform successive identical ad hoc queries.

Utilizing the Ancestor Field

We also compared execution times for queries that made use of the ancestor field construct with execution times for those that did not. As stated previously, the use of the ancestor field allows the composer of a query to "cheat" and shorten the traversal path in suitable circumstances. It was necessary to quantify the benefit obtained while recognizing that queries may not always be able to take advantage of this field.

```
insert into TEMP1 (ANCESTOR)
  select EAV_TEXT.ANCESTOR
  from EAV_TEXT
  where EAV_TEXT.VALUE="PSEUDOMONAS AERUGINOSA"
    and EAV_TEXT.ATTRIBUTE_ID=20;
insert into TEMP2 (VALUE) select EAV_OBJECTS.VALUE
  from TEMP1 inner join EAV_OBJECTS on TEMP1.ANCESTOR =
  EAV_OBJECTS.OBJECT_ID;
select distinct EAV_TEXT.VALUE
  from TEMP2 inner join EAV_TEXT on TEMP2.VALUE =
  EAV_TEXT.OBJECT_ID
  where EAV_TEXT.ATTRIBUTE_ID=9;
truncate table TEMP1; truncate table TEMP2;
```

Figure 6 Database query developed using previously created and indexed tables to store interim data.

```
Insert into TEMP1 (ANCESTOR)
  select EAV_TEXT.ANCESTOR
  from EAV_TEXT
  where EAV_TEXT.VALUE="PSEUDOMONAS AERUGINOSA"
    and EAV_TEXT.ATTRIBUTE_ID=20;
create index ANCESTOR_IND on TEMP1(ANCESTOR);
insert into TEMP2 (VALUE) select EAV_OBJECTS.VALUE
  from TEMP1 inner join EAV_OBJECTS on TEMP1.ANCESTOR
  = EAV_OBJECTS.OBJECT_ID;
create index VALUE_IND on TEMP2(VALUE);
select distinct EAV_TEXT.VALUE
  from TEMP2 inner join EAV_TEXT on TEMP2.VALUE
  = EAV_TEXT.OBJECT_ID
  where EAV_TEXT.ATTRIBUTE_ID=9;
drop index TEMP1.ANCESTOR_IND; drop index
TEMP2.VALUE_IND;
truncate table TEMP1; truncate table TEMP2;
```

Figure 7 Database query developed using un-indexed tables to store interim data, which are indexed only after data are inserted into the tables.

Monitoring System Performance

Operating system performance was monitored with Windows NT performance monitor in conjunction with SQL Server Objects and Counters. We recorded the following parameters:

- The number of physical database page reads per second
- The number of disk reads per second
- The percentage of time the hard disk was busy with read/write activity
- The percentage of time the processor handled non-idle threads

Page read frequency is a surrogate measure of disk I/O (input/output) efficiency. A high percentage of hard disk activity and a large number of disk reads per second suggest a potential disk I/O bottleneck. Finally, a high degree of processor activity suggests a potential processor-related bottleneck.

We performed these evaluations for Query 1 during initial execution (when there was no data caching) and the fifth run of the sequence of five (where caching effects were presumed to be most significant). SQL Server 7 and Windows NT provide system measure-

ments every second; we averaged these values over the duration of the query's execution to obtain summary measurements.

Results and Interpretation

Comparing Different EAV/CR Query Strategies for Attribute-centered Queries

As stated earlier, there are at least four different strategies to use when performing attribute-centered queries of EAV/CR data. Prior to contrasting the EAV/CR and conventional schemas for different memory configurations and database sizes, we had to decide which of these strategies was most viable. We report the comparative benchmarks with each approach for Query 1 (finding all patients with cultures positive for *P. aeruginosa*).

- *Initial execution times.* The approach of developing multiple SQL statements to create temporary tables produced the quickest initial run (97.3 sec). The approach of creating a single large query took 108.5 sec, indicating that, even with a three-alias join, the time the optimizer takes to devise a plan is significant. (Disk activity was higher; this is expected because the DBMS optimizer must consult disk-based structures to make its decision.) The approach of multiple SQL statements reusing indexed temporary tables gave the longest time (126.2 sec), indicating that, at least in this particular case, pre-indexing reusable tables does not necessarily pay off. The approach of multiple SQL statements reusing initially un-indexed temporary tables took 114.7 sec. This was better than pre-indexing but not as good as de novo creation of temporary tables.
- *Cached execution times.* Here, the single-statement approach gave results that were dramatically better than with multiple statements (14.3 sec versus 86.6 sec for temporary table creation, 95.9 sec for indexed temporary table reuse, and 89.9 sec for un-indexed temporary table reuse). The operating system statistics showed zero disk activity for the single statement, indicating caching effects. We concluded that the DBMS does not efficiently cache batches of SQL statements; only the last-used statement appears guaranteed to be cached. In practice, as we have stated earlier, caching does not provide any real benefit for ad hoc queries, which are unlikely to be repeated twice in succession.

Similar results were observed when these strategies were applied to Query 2 (find all specimens with *S. pneumoniae* and *C. albicans*).

The time difference of 10 percent between the multiple-statement and single-statement approaches is not particularly dramatic. We hypothesized, however, that as the number of classes involved in a query increased, these differences would become increasingly pronounced. We tried out a new query, "Find all patients with cultures positive for P. aeruginosa that showed resistance to ceftazidime." (This query also involves the "Antimicrobial_Tested" class and "Antimicrobial Name" attribute.) The initial execution times for the multiple-statement and single-statement queries were 194 sec compared with 338 sec, respectively. In other words, using multiple statements almost halved execution time. (Our experience is in line with that reported in a well-known database article,[16] which stated that giant, "elegant" SQL statements are often too complex for the limited intelligence of a DBMS optimizer.)

Because of these results, we used the multiple statement approach, with creation of temporary tables, to benchmark EAV/CR query performance for the remaining analyses.

Comparing Performance of the EAV/CR and Conventional Representations for Entity-centered Queries

Queries on the conventional and EAV/CR schemas gave comparable execution times:

- Query 3: Find All Microbiology Tests for a Single Patient. Initial times ranged from 1 to 2 sec for the EAV/CR schema, compared with 0.9 to 1.4 sec for the conventional schema. (The smaller and larger numbers in each range apply to the "Small" and "Big" versions of each database.) The corresponding cached times reduced to 0.6 and 0.5 sec irrespective of database size.
- Query 4: Find All Antimicrobials and Sensitivities Tested for a Particular Culture. Initial times were 2.1 to 2.4 sec for the EAV/CR schema compared with 2.2 to 2.7 sec for the conventional, with cached timings for both EAV/CR and conventional schemas being 1.5 to 1.7 sec.

To summarize, increasing the volume of the data by a factor of four (in the "Small" versus "Big" databases) had little effect on entity-centered queries. This is expected, because for such queries, which use indexes well and return small amounts of data, search time tends to increase logarithmically rather than linearly with data size. The results indicate that, compared with the EAV/CR schema, the conventional schema may be marginally more efficient (if at all,

since results for Query 4 are inconclusive), but the differences are not large enough to be of major practical importance.

Comparing Performance of the EAV/CR and Conventional Representations for Attribute-centered Queries

- Query 1: Find All Patients with Cultures Positive for *P. aeruginosa* (Table 1). The conventional queries were three to five times faster than their EAV/CR counterparts in the initial run, and 2 to 12 times faster in the cached run. We analyze the results below.

Increasing database size and system memory had little influence on the ratio for the initial run. With the smallest database, greater available RAM improved caching of EAV/CR queries, but this effect was lost for the largest database.

Using the ancestor field in the EAV/CR query provided little or no benefit for the smallest database but reduced the time for the large database. Here, the benefits were most pronounced with 64 Mb of RAM (with time reduced by a third), but less so (only one tenth) as more RAM was added.

The number of page reads per second was consistently larger in the EAV/CR schema, particularly as the database increased in size. This suggests that physical disk I/O is a likely bottleneck in this query operating on the EAV/CR model.

- Query 2: Find All Specimens Positive for *S. pneumoniae* and *C. albicans* (Table 2). Here, the EAV/CR queries that used the "Ancestor" field were much faster than queries on the conventional schema:

Table 1 ■

Execution Times (in Seconds) for Query 1: "Find All Patients with Cultures Positive for *Pseudomonas aeruginosa*"

	EAV/CR-Small		Conv-Small	EAV/CR-Big		Conv-Big
	Anc	NoAnc		Anc	NoAnc	
64-Mb RAM:						
Initial query	29.5	29.5	8.2	97.6	147	29.2
Cached	21.6	20.2	3.6	90.6	140	10.8
128-Mb RAM:						
Initial query	30.2	28.9	8.2	97.3	108	26.2
Cached	7.8	11.6	3.7	98.4	98.4	10.2

NOTE: Times for EAV/CR schema are displayed with and without ancestor construct. Anc indicates ancestor; NoAnc, no ancestor.

Table 2 ■

Execution Times (in Seconds) for Query 2: "Find All Specimens Positive for Both *Pseudomonas aeruginosa* and *Candida albicans*"

	EAV/CR-Small		Conv-Small	EAV/CR-Big		Conv-Big
	Anc	NoAnc		Anc	NoAnc	
64-Mb RAM:						
Initial query	3.3	21.3	6.9	14.3	141	25.3
Cached	0.7	13.6	4.5	1.5	139	11.0
128-Mb RAM:						
Initial query	3.7	22.0	6.8	12.9	83.5	25.1
Cached	0.7	13.7	4.5	1.4	49.4	11.2

NOTE: Times for EAV/CR schema are displayed with and without ancestor construct. Anc indicates ancestor; NoAnc, no ancestor.

twice as fast with the initial query, six to eight times as fast when cached. Changing database size or increasing system memory had little effect on these ratios.

When the ancestor field was not used, the EAV/CR query took longer, with the difference in execution times widening for initial runs as database size increased. With 64 MB of RAM allocated, the ratios were 3 for the small databases and 5.6 for the large databases. As with Query 1, however, the ratios decreased (to 3.2 and 3.3, respectively) when 128 Mb of RAM was allocated.

Our results indicate that while EAV/CR is generally less efficient than its conventional equivalent, the inefficiency can be mitigated, at least in part, by additional memory. The dramatic benefits of the ancestor field (specimen ID) in Query 2, which uses this field specifically, should be put in perspective. Specifically, incorporation of this extra field in the schema amounts to deliberate denormalization, a standard space-for-time tradeoff.* (We could also have done this in the conventional schema, by adding a specimen ID column redundantly to the cultures and antimicrobial sensitivity tables. In this case, the altered conventional schema would almost certainly have outperformed the EAV/CR schema considerably.)

Our results do suggest that, if performance for certain types of commonly executed queries is critical in a particular database, judicious use of ancestor fields can exert more of an impact than the schema's un-

**Normalization* refers to the database design principle of minimizing data redundancy by storing a fact, as far as possible, in only a single place. *De-normalization* refers to design that violates one or more of these principles.

derlying design (i.e., EAV/CR or conventional). In many electronic patient record systems, for example, almost every item of data is tagged with the ID of the patient to whom it directly or indirectly pertains, because a very large portion of queries relates to individual patients.

Discussion

We performed a pilot exploration of EAV/CR database efficiency issues that can help guide designers and programmers of EAV/CR databases, such as electronic patient record systems, as to the performance penalty they could expect if they tried to execute complex queries on the EAV/CR data directly, rather than exporting subsets of the data to a conventional database for querying purposes. However, the following limitations of our study must be noted.

We examined performance monitors for a single database engine running on a single operating system platform. We chose this configuration largely because of the available built-in system monitoring tools. The results generated here may not completely map to other database engines and operating systems. (Identical queries conducted on the two schemas using a Microsoft Access 97 database engine running on Windows 98, however, yielded qualitatively similar results.)

We cannot predict how performance would be affected by increasing the database to sizes larger than those tested here. Our results suggest, however, that the difference in performance appears to widen as database size increases for certain queries and that this can be offset, at least in part, by adding system memory.

We did not make use of the parallel processing capabilities of the SQL Server database engine, since we employed a single processor machine for testing.

Investigations are under way to study additional methods to optimize query generation and execution.

Our databases were populated with data from one specific domain, microbiology. We cannot predict how performance times would differ in a database populated from increasingly heterogeneous sources of data. We would expect, however, that the effort needed to develop queries would increase disproportionately in the conventional schema compared with the EAV/CR schema.

The logical schema of our test database may not be the best candidate for demonstrating the full potential value for an EAV/CR schema, since the data do not exhibit sparseness.

The Windows NT operating system performs certain background tasks for system maintenance that could conceivably affect specific performance runs. We tried to minimize these effects by generously dedicating 128 Mb of physical RAM to the operating system at all times.

These are the lessons we learned from this exercise:

- For entity-centered queries, EAV/CR schemas are about as efficient as their conventional counterparts. Attribute-centered queries are distinctly slower for EAV/CR data than for conventional data (three to five times slower, for our test data). Adding more memory improves the ratio, especially for large volumes of data, as shown by performance monitor statistics.
- The strategy of querying an EAV/CR database with simple statements run sequentially is a viable one. Each individual statement is simple to understand; in fact, it is simple enough to be created through a query generator that has knowledge of the EAV/CR logical schema. (A query generator for ACT/DB, an EAV/CR database for clinical studies, is described in Nadkarni and Brandt.[9]) We also found that using this query approach was more efficient than creating a single, large SQL query statement. The difference in efficiencies appeared to increase as queries became more complex. Additional studies are under way to examine this approach under a wider range of conditions.
- The "ancestor" field yielded impressive benefits in improving efficiency for certain queries. Its primary limitations are that not all queries can make use of it and that certain domains may not possess "natural" candidate ancestor fields. (The existence of an ancestor field might also complicate the design of an automatic query generator considerably, because the generator must know the circumstances in which shortcuts can be taken.)

While attribute-centered EAV/CR queries are slower than queries on conventional schemas, this does not disqualify EAV/CR schemas from consideration for warehousing biomedical data. For a complex schema, the increased execution times are significantly offset by ease of database and query maintainability in the EAV/CR model. The conventional query's speed advantage (being three to five times faster) may appear to be discouraging for EAV/CR, but the fact is that the longest EAV/CR query in our fairly large data set took less than 3.5 min to complete. Continuing enhancements of CPU speed, increasing RAM capacity, and the availability of affordable multiprocessor machines

will lower absolute times further, even though they would benefit conventional and EAV schemas equally.

In any case, long query execution times tend to be far less critical for non–real-time, attribute-centered queries, which are often submitted in batch mode. (Patient-centered queries typically submitted in an electronic patient record system, in contrast, demand quick response time.) Perhaps the most compelling justification for moving EAV/CR data subsets to specially designed, conventionally structured data marts is when users who demand response times of a few seconds repeatedly query these subsets.

Conclusions

This paper describes a pilot project to explore issues in query performance for EAV/CR and conventional database representations. Although we found that attribute-centered queries performed less efficiently in the EAV/CR model, we feel that many of the benefits inherent in the EAV/CR representation help offset this decrease in performance. Purchasing more memory, additional processors, or faster hardware , or a combination of these, may prove a very cost-effective approach to handling these potential inefficiencies, particularly if the alternative is to maintain two parallel versions of a database with different structures. We plan to continue building and optimizing query strategies to support this representation as well as testing its ability to handle increasingly complex databases for alternative biomedical domains.

References ■

1. Winston PH. Artificial Intelligence. 2nd ed. Reading, Mass: Addison-Wesley, 1984.
2. Huff SM, Berthelsen CL, Pryor TA, Dudley AS. Evaluation of an SQL model of the HELP patient database. Proc 15th Symp Comput Appl Med Care. 1991:386–90.
3. Huff SM, Haug DJ, Stevens LE, Dupont RC, Pryor TA. HELP the next generation: a new client–server architecture. Proc 18th Symp Comput Appl Med Care. 1994:271–5.
4. Friedman C, Hripcsak G, Johnson S, Cimino J, Clayton P. A generalized relational schema for an integrated clinical patient database. Proc 14th Symp Comput Appl Med Care. 1990:335–9.
5. Johnson S, Cimino J, Friedman C, Hripcsak G, Clayton P. Using metadata to integrate medical knowledge in a clinical information system. Proc 14th Symp Comput Appl Med Care. 1990:340–4.
6. Nadkarni PM, Marenco L, Chen R, Skoufos E, Shepherd G, Miller P. Organization of heterogeneous scientific data using the EAV/CR representation. J Am Med Inform Assoc. 1999; 6(6):478–93.
7. Niedner CD. The entity-attribute-value data model in radiology informatics. Proceedings of the 10th Conference on Computer Applications in Radiology. Anaheim, Calif: Symposia Foundation, 1990:50–60.
8. Nadkarni PM, Brandt C, Frawley S, et al. Managing attribute-value clinical trials data using the ACT/DB client–server database system. J Am Med Inform Assoc. 1998;5(2): 139–51.
9. Nadkarni P, Brandt C. Data extraction and ad hoc query of an entity-attribute-value database. J Am Med Inform Assoc. 1998;5(6):511–27.
10. Nussbaum BE. Issues in the design of a clinical microbiology database within an integrated hospital information system. Proc 15th Annu Symp Comput Appl Med Care. 1991: 328–32.
11. Delorme J, Cournoyer G. Computer system for a hospital microbiology laboratory. Am J Clin Pathol. 1980;74:51–60.
12. Department of Veterans Affairs. Decentralized Hospital Computer System Version 2.1: Programmer's Manual. San Francisco, Calif: Information Systems Center, 1994.
13. Celko J. SQL for Smarties: Techniques for Advanced SQL Programmers. San Mateo, Calif: Morgan Kaufman, 1996.
14. Sybase Corporation. Adaptive Server Anywhere (ASA) New Features and Upgrading Guide. (Chapter 4. Query Optimization Enhancements.) Available at: http://sybooks.sybase.come:7000/onlinebooks/group-aw/awg0603e/dbupen6/. Accessed Jul 29, 2000.
15. Zaniolo C, Ceri S, Faloutsos C, Snodgrass R, Subrahmanian V, Zicari R. Advanced Database Systems. San Francisco, Calif: Morgan Kaufmann, 1997.
16. Celko J. Everything you know is wrong. DBMS Mag. 1996; 9(9):18–20.
17. Wang P, Pryor TA, Narus S, Hardman R, Deavila M. The Web-enabled IHC enterprise data warehouse for clinical process improvement and outcomes measurement. AMIA Annu Fall Symp. 1997:1028.