(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
23 December 2010 (23.12.2010)

PCT

(10) International Publication Number
WO 2010/147950 A2

(51) International Patent Classification: Not classified

(21) International Application Number:
PCT/US2010/038622

(22) International Filing Date:
15 June 2010 (15.06.2010)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
61/187,584 16 June 2009 (16.06.2009) US
61/226,430 17 July 2009 (17.07.2009) US

(72) Inventor; and
(71) Applicant *(for all designated States except US)*: **COHEN, Jonathan** [US/US]; 4761 Fernridge Lane, Mercer Island, WA 98040 (US).

(74) Agent: **PHILIPP, Adam, L.k.**; Aeon Law, 1525 4th Ave., Suite 800, Seattle, WA 98101 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PE, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *without international search report and to be republished upon receipt of that report (Rule 48.2(g))*

(54) **Title**: DATA VISUALIZATION SYSTEM AND METHOD

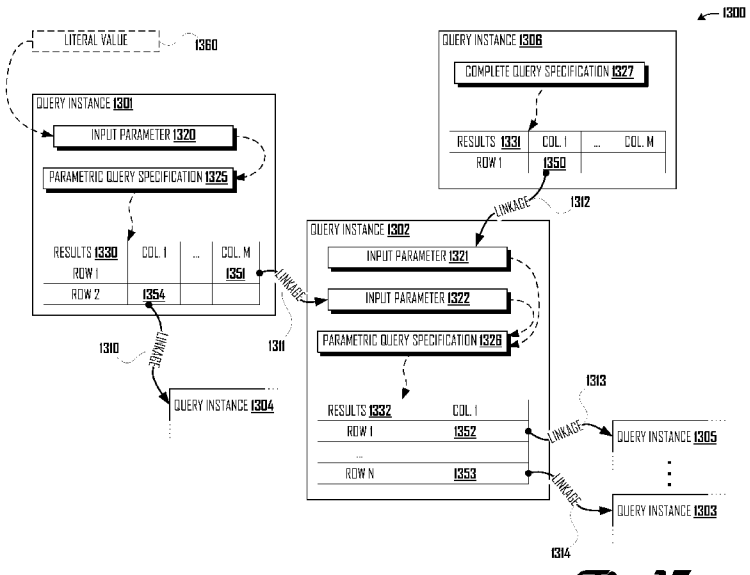

Fig. 13

(57) **Abstract**: A data visualization system and method are provided herein.

WO 2010/147950 A2

DATA VISUALIZATION SYSTEM AND METHOD

CROSS REFERENCE TO RELATED APPLICATIONS

[Para 01] This application claims the benefit of priority to Provisional Application No. 61/187,584, filed June 16, 2009, titled "Data Visualization System and Method," having Attorney Docket No. VSEY-2009002, and naming inventor Jonathan Cohen. This application also claims the benefit of priority to Provisional Application No. 61/226,430, filed July 17, 2009, titled "Heirarchical Diagram System and Method," having Attorney Docket No. VSEY-2009003, and naming inventor Jonathan Cohen. The above-cited applications are incorporated herein by reference in their entireties, for all purposes.

[Para 02] Provisional Application No. 61/226,430, mentioned above, is also included as Appendix A to this application. Appendix A discloses systems and methods for constructing diagrams of deeply hierarchical data such as may be used to implement one embodiment of the data visualization system disclosed herein.

FIELD

[Para 03] The present disclosure relates to electronic databases, and, more particularly, to systems and methods for visualizing relationships among data in a database.

BACKGROUND

[Para 04] People often want to select, view, and/or interact with data in a database in a visual, structured manner. Various programs, such as Microsoft Excel, from Microsoft Corporation of Redmond Washington, allow a user to show data in a statistical manner, but that type of graph does not allow the viewer to see the structure of the data within the database.

[Para 05] In other cases, the visualization does show the structure, but in a relatively flat way. For example, the data visualization tool Many Eyes, provided by International Business Machines Corporation of Armonk NY, provides several types of visualizations, including a "network diagram," which depicts an interconnected graph of nodes. For

another example, the graphing tool Gruff, provided by Franz Inc. of Oakland, CA, allows a user to enter a standard, complete textual query and place the results in a graph. The user can click on a graphed result and see the objects from the database that are related to that result via predefined relationships.

[Para 06] In some cases, it is possible to create more detailed visualizations, for example, by manually constructing a diagram, using a drawing software program, and/or writing special-purpose software to obtain a particular set of data and layout a graph. For example, a software program such as Visio, provided by Microsoft Corporation of Redmond WA, may enable a user to manually lay out a graph of objects with object labels populated with data from a data source, such as a database. There are also products that provide database graphing and report generation facilities.

[Para 07] However, existing solutions may fail to enable a user to easily explore the structure of the data in relational database, including following particular items of data and/or comparing various branching pathways throughout various tables in a database or set of databases.

BRIEF DESCRIPTION OF THE DRAWINGS

[Para 08] Figure 1 is a system diagram showing a number of interconnected devices in accordance with one embodiment.

[Para 09] Figure 2 is a block diagram of a device that provides an exemplary operating environment for various embodiments.

[Para 10] Figure 3 illustrates an exemplary data visualization user interface in accordance with one embodiment.

[Para 11] Figure 4 illustrates an exemplary query pane user interface in accordance with one embodiment.

[Para 12] Figure 5 illustrates an exemplary linkage pane user interface in accordance with one embodiment.

[Para 13] Figure 6 illustrates a number of exemplary queries in accordance with one embodiment.

[Para 14] Figure 7 illustrates a number of exemplary linkages in accordance with one embodiment.

[Para 15] Figure 8 illustrates an exemplary visualization graph in accordance with one embodiment.

[Para 16] Figure 9 illustrates an exemplary data visualization routine in accordance with one embodiment.

[Para 17] Figure 10 illustrates an exemplary reusable static or parametric query definition subroutine in accordance with one embodiment.

[Para 18] Figure 11 illustrates an exemplary query instance execution subroutine in accordance with one embodiment.

[Para 19] Figure 12 illustrates an exemplary query explosion subroutine in accordance with one embodiment.

[Para 20] Figure 13 illustrates a conceptual reusable parametric query visualization graph in accordance with one embodiment.

DESCRIPTION

[Para 21] The detailed description that follows is represented largely in terms of processes and symbolic representations of operations by conventional computer components, including a processor, memory storage devices for the processor, connected display devices and input devices. Furthermore, these processes and operations may utilize conventional computer components in a heterogeneous distributed computing environment, including remote file servers, computer servers, and/or memory storage devices. Each of these conventional distributed computing components is accessible by the processor via a communication network.

[Para 22] The phrases "in one embodiment," "in various embodiments," "in some embodiments," and the like are used repeatedly. Such phrases do not necessarily refer to the same embodiment. The terms "comprising," "having," and "including" are synonymous, unless the context dictates otherwise.

[Para 23] Reference is now made in detail to the description of the embodiments as illustrated in the drawings. While embodiments are described in connection with the drawings and related descriptions, there is no intent to limit the scope to the embodiments disclosed herein. On the contrary, the intent is to cover all alternatives, modifications and equivalents. In alternate embodiments, additional devices, or combinations of illustrated

devices, may be added to, or combined, without limiting the scope to the embodiments disclosed herein.

[Para 24] Various embodiments of a data visualization system, as disclosed herein, may display data from a database as a visualization graph of parametric query nodes and links. In one embodiment, instances of reusable query objects specify result nodes in the visualization graph, and instances of linkage objects specify links between query instances in the result graph. In some embodiments, each node in the visualization graph may be an instance of a unique query object. In other embodiments, more than one node may be an instance of a single query object. Similarly, linkage instances in the result graph may or may not share linkage objects.

[Para 25] For example, Figure 13 conceptually illustrates a visualization graph 1300 in accordance with one embodiment. Nodes in graph 1300 are query instances 1301-1306, and edges in graph 1300 are linkages 1310-1314.

[Para 26] Each of query instances 1301-1306 is an instance of a particular reusable query object (not shown). Every query object comprises a query specification (either parametric or complete) and one or more identified output columns. A complete query specification represents a database query that is fully defined and may be executed without additional data. For example, a complete query specification might represent a query such as "SELECT FOO FROM BAR WHERE BAT = 3". A query object having a complete query specification is referred to herein as a "static query," and instances thereof are referred to as "static query instances." For example, static query instance 1306 includes a complete query specification 1327, which can be executed as-is to obtain result rows 1331.

[Para 27] By contrast, a parametric query specification includes one or more input parameters. For example, a parametric query specification might represent a query such as "SELECT FOO FROM BAR WHERE BAT = [-]", where [-] indicates an input parameter, whose value must be supplied from a linkage or from another source before the query can be executed. A query object having a parametric query specification is referred to herein as a "parametric query," and instances thereof are referred to as "parametric query instances." Query instances 1301-1305 are parametric query instances.

[Para 28] In some cases, a parametric query instance may obtain a literal value for an input parameter. Some embodiments may provide a graphical control (not shown, but see Figure 8, discussed below) with which a user may type, select, or otherwise enter a number or string to be provided as an input parameter to a parametric query instance. The parametric query instance may then substitute the provided number or string for the input parameter in the parametric query specification and execute the query. For example, parametric query instance 1301 obtains literal value 1360 for input parameter 1320 to complete parametric query specification 1325, which can then be executed to obtain result rows 1330.

[Para 29] More pertinently, in other cases, a parametric query instance may obtain a value for an input parameter via a linkage, such as linkages 1310-1314. Put simply, a linkage maps an output column of a source query object (which may be a static query or a parametric query) directly to an input parameter of a destination query object (which can only be a parametric query). For example, parametric query instance 1302 obtains a value for input parameter 1321 via linkage 1312 from result field 1350 of static query instance 1306, and parametric query instance 1302 obtains a value for input parameter 1322 via linkage 1311 from result field 1351 of static query instance 1301. Using the provided values, parametric query specification 1326 can be completed and executed to obtain result rows 1332. Similarly, linkages 1310 and 1313-1314 respectively provide values from fields 1354 and 1352-1353 to input parameters (not shown) of parametric query instances 1303-1305.

[Para 30] In some cases, reusable linkage objects may be defined and individually instantiated for each linked query instance pair. For example, linkages 1313 and 1314 may be individual instantiations of the same reusable linkage object (not shown). In other cases, singleton linkages may be defined for a particular linked query instance pair.

[Para 31] Thus, visualization graph 1300, with its query instance nodes and linkage edges, may provide a visualization of relationships between various data records. A visualized data record may come from a single table, from multiple tables, and/or from tables in multiple databases. There are typically many relationships between data records in a database, and various embodiments may facilitate flexible selection of particular relationships for exploration and/or visualization.

[Para 32] In some embodiments of a data visualization system, an icon-based and/or point-and-click mechanism may enable a user to identify, visualize, and/or explore data relationships of interest without requiring the user to read or write Structured Query Language ("SQL") or other textual database computer language.

[Para 33] Figure 1 illustrates a number of interconnected devices in accordance with one embodiment. Data visualization device 200 is connected to one or more local and/or remote databases 105. In various embodiments, database 105 may comprise a database management system ("DBMS"). In an exemplary embodiment, database 105 may comprise a relational database management system ("RDBMS") database such as Oracle Database, provided by Oracle Corporation of Redwood Shores California; Microsoft SQL Server, provided by Microsoft Corporation of Redmond Washington; MySQL provided by MySQL AB of Uppsala Sweden and Cupertino California; and the like. In other embodiments, alternate DBMS may also be used, such as an object database ("ODBMS"), column-oriented DBMS, correlation database DBMS, and the like.

[Para 34] Figure 2 illustrates several components of an exemplary data visualization device 200. In some embodiments, data visualization device 200 may include many more components than those shown in Figure 2. However, it is not necessary that all of these generally conventional components be shown in order to disclose an illustrative embodiment. As shown in Figure 2, data visualization device 200 includes a database interface 230 for connecting to database 105. Database interface 230 includes the necessary circuitry, drivers, authentication credentials, and/or program code for such a connection and is constructed for use with an appropriate protocol. In some embodiments, database interface 230 may comprise a network interface.

[Para 35] Data visualization device 200 also includes a processing unit 210, a memory 250, and a display 240, all interconnected, along with database interface 230, via bus 220. Memory 250 generally comprises a random access memory ("RAM"), a read only memory ("ROM"), and/or a permanent mass storage device, such as a disk drive. Memory 250 stores program code for reusable parametric query visualization routine 900, reusable static or parametric query definition subroutine 1000, query instance execution subroutine 1100, and reusable parametric query explosion subroutine 1200. In addition, memory 250 also stores an operating system 255.

[Para 36] In some embodiments, memory 250 also includes one or more session files 265, which may be used to save a user's work. In one embodiment, a saved session may be organized into one or more files contained in a directory. Some embodiments may provide a user interface for managing, sharing, and/or reusing queries, linkages, and/or result graphs stored in a session file or files.

[Para 37] These and other software components may be loaded from a computer readable storage medium 295 into memory 250 of data visualization device 200 using a drive mechanism (not shown) associated with a non-transient computer readable storage medium 295, such as a floppy disc, tape, DVD/CD-ROM and other optical media, memory card, and the like. In some embodiments, software components may also be loaded via a network interface (not shown) or other non-storage media.

[Para 38] Figure 3 illustrates an exemplary data visualization user interface 300 in accordance with one embodiment. Data visualization user interface 300 includes control pane 305, which in some embodiments may display a list of object types, objects, and the like. In one embodiment. detail pane 310 may display details related to the active object, as well as a field to enter notes, a description, and/or other information associated with the active object.

[Para 39] Data visualization user interface 300 also includes error pane 320, which in one embodiment, displays a descriptive list of errors that may have occurred (if any). In some embodiments, when an error-causing condition is corrected, the corresponding error display may be removed.

[Para 40] Data visualization user interface 300 includes query pane 330. An exemplary query pane is illustrated in Figure 4 and discussed below in greater detail. Data visualization user interface 300 includes linkage pane 325. An exemplary linkage pane is illustrated in Figure 5 and discussed below in greater detail. Data visualization user interface 300 also includes result pane 315. An exemplary result pane is illustrated in Figure 8 and discussed below in greater detail.

[Para 41] Figure 4 illustrates an exemplary query pane user interface 400 in accordance with one embodiment. In some embodiments, a query pane may display a graph with two kinds of vertices: table vertices (e.g., vertices 434, 435) and parameter

vertices (e.g. vertex 440). The displayed query graph may visually indicate a complete or partial query specification of a reusable static or parametric query.

[Para 42] For example, the illustrated query pane 400 includes table vertices 434 and 435, each of which includes a table control 410, 415 by which a user may specify which database table(s) the reusable parametric query will access (e.g. from database 105). Table vertices 434, 435 also include one or more column controls 420, 425, 430, from which a user may select one or more columns to be retrieved (e.g. from database 105) when the query is run.

[Para 43] In one embodiment, a query graph that includes more than one table vertex (e.g., vertices 434, 435) represents a query specification including a JOIN operation. In one embodiment, such a join operation is visually indicated via a join control 405 connecting table nodes 434 and 435. In one embodiment, an inner join is performed by default, but a user may specify alternate joins using a join definition user interface (not shown).

[Para 44] In one embodiment, table vertices 434, 435 may also include one or more "nickname" fields (not shown), by which a user may provide descriptive names for tables and/or columns. In some embodiments, a query object may further comprise formatting information for columns of data returned by the query. In some embodiments, a query object may further comprise default query and/or linkage information to facilitate explosion operations, as discussed below.

[Para 45] Query pane 400 also includes a parameter vertex 440. In one embodiment, parameter vertices specify which rows from the indicated table(s) will be included in the result graph. In one embodiment, a user can enter a static literal value, such as a customer number, into a parameter vertex 440. The user may further create a parameter link 445 to connect the parameter node (e.g. 440) to one of the indicated columns (e.g. 425). In one embodiment, a parameter link 445 represents by default an Equals condition between the columns, but a user may select other relations conditions using a parameter definition user interface (not shown).

[Para 46] In some embodiments, in addition to merely holding a static value, parameter vertex 440 can also hold an arbitrary query language statement. In other embodiments, parameter vertex 440 can represent an "input parameter." As discussed

above, an input parameter stands in place of a value that is required by a partial query specification to return a value. In various embodiments, a user can provide a static literal parameter value in a visualization graph, and/or a parameter value can be obtained from another query node.

[Para 47] Figure 5 illustrates an exemplary linkage pane user interface 500 in accordance with one embodiment. In one embodiment, linkage definition interface 505 defines or maps how to pass selected values from the result column of a particular source query (indicated by source query control 550) to a particular input parameter of a particular destination query (indicated by destination query control 555). In some embodiments, when a user creates a new linkage, one or both of the query controls 510, 515 may be pre-populated with contextually-determined likely query identifiers. If the pre-populated query identifiers are not what the user intended, he or she may select other queries using query controls 510, 515.

[Para 48] In some embodiments, a linkage 505 could define more than one source query to support gathering values from multiple source results. In some embodiments, a defined linkage may be re-used in many different places in a result graph.

[Para 49] Source query control 550 displays one or more source query output columns 520, 525, 530, 535, one or more of which may be used as data sources for an input parameter of the selected destination query 555. Similarly, destination query control 555 displays one or more input parameters 540 of the selected destination query. To complete the linkage specification, a user uses a link control 560 to specify a source output column 535 and a destination input parameter 540. In one embodiment, link control 560 may be created by dragging with a pointing device from a source column drag source 545 to a destination column drag sink 565. Alternatively, in some embodiments, the user may enter a value for some or all destination query parameters 540.

[Para 50] In some embodiments, a user may further specify an operation to be performed with data passing via the defined linkage. For example, a user may specify that data passing via a linkage be converted from one unit of measure to another and/or be tested for a validity condition. In some embodiments, a linkage may further be specified

to warn a user if, for example, the linked source result values are not appropriate for the destination query parameter.

[Para 51] Figure 6 illustrates query definition panes 605, 610, 615 corresponding to parametric query instances illustrated in Figure 8. Figure 7 illustrates linkage definition panes 705, 710 corresponding to linkages illustrated in Figure 8.

[Para 52] Figure 8 illustrates an exemplary visualization graph 800, such as may be displayed in result pane 315, in accordance with one embodiment. In the illustrated embodiment, objects defined in query definition panes 605, 610, 615 and linkage definition panes 705, 710 provide information necessary to obtain a visualization graph 800. In some embodiments, visualization graph 800 comprises one or more query nodes 840-843 connected by linkages indicated by linkage controls 850-852. Each of query nodes 840-843 corresponds to an instance of a parametric query (see query definition panes 605, 610, 615, illustrated in Figure 6).

[Para 53] In the illustrated visualization graph 800, query nodes 840-843 are labeled with parenthetical descriptive identifiers merely to help clarify the description of visualization graph 800. For example, query node 841 is parenthetically identified as a "parent" node, which reflects its relationship to the "explosion child" nodes 842-843. (In some embodiments, an "explosion" routine, e.g. routine 1200 as illustrated in Figure 12 and discussed below, may automatically create "explosion child" nodes such as nodes 842-843.) Similarly, query node 840 is parenthetically identified as a "specification" node, which reflects its relationship to "parent" node 841 (namely that result field 855 provides a value for input parameter 826 of node 841, thereby completing the specification of parametric query 811). However, these descriptive identifiers do not limit the functional roles that may be played by nodes 840-843. For example, via further linkage and/or explosion operations (not shown), "child" nodes 843 and/or 843 could further function as "parent" nodes to one or more explosion children nodes (not shown).

[Para 54] In the illustrated embodiment, query nodes 840-843 include graphical query selection controls 815-818, by which a user can select a static or parametric query to instantiate for the node via a query selection interface (not shown, e.g., a drop-down menu, text entry field, or the like). As illustrated in Figure 8, node 840 corresponds to an instance of parametric query 810, node 841 corresponds to an instance of parametric

query 811, node 842 corresponds to an instance of parametric query 812, and node 843 corresponds to an instance of parametric query 813. In one embodiment, when a new static or parametric query is selected for a given node, visualization graph 800 may automatically update to reflect the new query selection.

[Para 55] In the illustrated embodiment, query nodes 840-843 include graphical input parameter controls 825-828, by which a user can specify one or more values for the node's parametric query instance. For example, using graphical input parameter control 825, a user has specified a literal string value ("Robert Williams") for the CUSTOMER_NAME input parameter of the parametric query instance to which node 840 corresponds. Values for the input parameters in nodes 841-843 are specified via linkages, which are graphically indicated by linkage controls 850-852, respectively. Therefore, in the illustrated embodiment, graphical input parameter controls 826-828 display the current value provided via the indicated linkage. In some embodiments, a user may indicate, define, and/or specify a linkage using a graphical connection control (e.g. connection controls 860-865) corresponding to a result row. Once a linkage has been specified, in some embodiments, a user may edit and/or change a linkage using an interface (not shown) accessed via linkage controls 850-852.

[Para 56] For a node corresponding to a parametric query node (e.g. nodes 840-843), result rows (e.g. result rows 830-835) can be obtained once values are provided for all of the parametric query instance's input parameters to complete the instance's partial query specification. Each result row (e.g. result rows 830-835) includes one or more values for one or more fields corresponding to one or more output columns (e.g. output columns 870-879) defined in the corresponding static or parametric query. In other embodiments, a node may graphically depict is result rows in a non-tabular format.

[Para 57] In various embodiments, visualization graph 800 and/or any of nodes 840-843 may include additional controls (not shown). For example, In one embodiment, a graphical node may include one or more display/hide controls (not shown) to enable a user to selectively determine how much information is displayed in the node, which may enable to user to hide details that he or she does not wish to focus on and/or to save space in visualization graph 800.

[Para 58] Thus, as illustrated in Figures 3-8 and discussed above, various embodiments may be utilized to generate visualization graphs from database records. Concomitantly, various user interface tools may facilitate providing correct data to queries and interchanging various queries, parameters, and/or linkages to enable a user to interactively investigate and/or explore structured data.

[Para 59] Figure 9 illustrates one possible exemplary flow of a portion of a visualization routine 900, such as may be used to create a visualization graph (e.g. graph 800) in accordance with one embodiment. In various embodiments, the illustrated steps may be performed in a different order than indicated and/or similar steps may be added (or omitted), according to directions received from a user.

[Para 60] In subroutine block 1000A, routine 900 defines a first static or parametric query according to subroutine 1000 (see Figure 10). In block 905, routine creates an instance of the first static or parametric query. In subroutine block 1100A, routine 900 obtains one or more result rows for the first static or parametric query instance according to subroutine 1100 (see Figure 11). In block 915, routine 900 displays a graphical query node corresponding to the first static or parametric query instance. For example, in block 915, routine 900 may display a query node such as node 840 in Figure 8, discussed above.

[Para 61] In subroutine block 1000B, routine 900 defines a second parametric query according to subroutine 1000 (see Figure 10, discussed below). In block 905, routine 900 creates an instance of the second parametric query. In block 925, routine 900 links an input parameter of the second parametric query instance to an output column of the first static or parametric query instance. In some embodiments, linking these query instances may include defining and instantiating a reusable linkage object mapping an input parameter of the second parametric query to an output column of the first static or parametric query.

[Para 62] In subroutine block 1100B, routine 900 obtains one or more result rows for the second parametric query instance according to subroutine 1100 (see Figure 11, discussed below). In block 930, routine 900 displays a graphical query node corresponding to the second parametric query instance. For example, in block 930, routine 900 may display a query node such as node 841 in Figure 8, discussed above. In

some embodiments, routine 900 may also display a linkage control, such as linkage control 850 (see Figure 8, discussed above), indicating the link between the input parameter of the second parametric query instance and the output column of the first static or parametric query instance.

[Para 63] In block 1200, routine 900 performs an explosion routine, such as routine 1200, illustrated in Figure 12 and discussed below, to automatically create "child" or "explosion" nodes corresponding to the one or more result rows for the second parametric query instance obtained in block 1100B. For example, in block 1200, routine 900 may automatically create and display one or more explosion query nodes, such as nodes 842-843 in Figure 8, discussed above. In some embodiments, routine 900 may also automatically display one or more linkage controls, such as linkage controls 851-852 (see Figure 8, discussed above), indicating the link between explosion query nodes and the second parametric query node. In various embodiments, the user may continue drilling down into the data thus revealed and/or may otherwise continue to explore the data.

[Para 64] Figure 10 illustrates an exemplary subroutine 1000 for defining a reusable static or parametric query in accordance with one embodiment. The blocks illustrated in Figure 10 represent one possible sequence of operations comprising subroutine 1000, presented herein for clarity. However, in many embodiments, some or all of the illustrated operations may take place out of the illustrated sequence, possibly in response to directions from a user.

[Para 65] In block 1005, subroutine 1000 obtains a query specification for the reusable static or parametric query object being defined. For example, in one embodiment, a graphical query definition pane such as pane 400 (see Figure 4, discussed above) may be employed to obtain the query specification.

[Para 66] In decision block 1010, subroutine 1000 determines whether the obtained query specification is complete or partial. As discussed above, a complete query specification represents a database query that is fully defined and may be executed without additional data. If the obtained query specification is complete, then the query object being defined is a static query specification, and routine 900 skips to block 1055, discussed below.

[Para 67] By contrast, a partial query specification requires one or more additional values before the query can be executed. If the obtained query specification is partial, then the query object being defined is a parametric query specification, and routine 900 proceeds to block 1015, in which one or more input parameters required to complete the partial query specification are defined. For example, in one embodiment, using a graphical query definition pane such as pane 400, a user may define an input parameter such as input parameter 440 (see Figure 4, discussed above).

[Para 68] In block 1055, subroutine 1000 identifies output columns in the query specification. For example, in one embodiment, using a graphical query definition pane such as pane 400, output columns 420 and 425 may be identified (see Figure 4, discussed above). Beginning in block 1060, subroutine 1000 processes each output column. In block 1065, subroutine 1000 optionally defines a default explosion query and/or default explosion linkage for the current output column. If no default is set for an output column, a user may be prompted to determine an appropriate linkage and/or query when an explosion operation is performed on an instance of the query object being defined. In ending loop block 1070, subroutine 1000 iterates back to block 1060 to process the next output column (if any). Subroutine 1000 ends in block 1099.

[Para 69] Figure 11 illustrates an exemplary execution subroutine 1100 for a subject query instance in accordance with one embodiment. In decision block 1105, subroutine 1100 determines whether a current set of result rows for the subject query instance already exist and/or have been cached. If so, subroutine 1100 skips to block 1199, returning the result rows. If not, in decision block 1110, subroutine 1100 determines whether the subject query instance is a static query instance. If so, subroutine 1100 skips to block 1145 to execute the static query instance's fully-specified or complete query.

[Para 70] If in decision block 1110, subroutine 1100 determines that the subject query instance is a parametric query instance, then beginning in loop block 1115, subroutine 1100 processes each of the subject query instance's input parameters.

[Para 71] In block 1120, subroutine 1100 determines whether the current input parameter is linked to a specification query instance, which will provide an input value

according to one of its output fields. In some embodiments, this determination may involve prompting the user to provide a linkage or to provide a literal input value.

[Para 72] If in decision block 1120, subroutine 1100 determines that a value for the current input parameter is not provided by a link to a specification query instance, then in block 1125, subroutine 1100 obtains a literal value for the input parameter. For example, in some embodiments, subroutine 1100 may prompt the user to enter and/or select a literal value for the parameter.

[Para 73] On the other hand, if in decision block 1120, subroutine 1100 determines that a value for the current input parameter is to be provided by a link to a specification query instance, then in block 1135, subroutine 1100 obtains an input value from the linked field of the specification query instance. In some embodiments, obtaining such a linked value may include transforming the value in the linked field. For a simple example, the linked value may be transformed from one unit to another unit before being provided to the subject query instance as the current input parameter.

[Para 74] In ending loop block 1140, subroutine 1100 loops back to block 1115 to process the next input parameter (if any). Once all input parameters have been processed, the parametric query's partial query specification can be completed and executed to obtain a set of result rows, which are returned to the caller when subroutine 1100 ends in block 1199. In some embodiments, subroutine 1100 may additionally perform other operations, such as caching the result rows for re-use, for example.

[Para 75] Figure 12 illustrates an exemplary subroutine 1200 for exploding a subject query instance in accordance with one embodiment. Beginning in loop block 1205, subroutine 1200 processes each result row of the subject query instance in turn. (It will be appreciated that in some embodiments, subroutine 1200 may only process a user-indicated subset of the result rows of the subject query instance.)

[Para 76] In block 1210, subroutine 1200 instantiates a determined "explosion" parametric query for use in the explosion operation. In some embodiments, the explosion parametric query may be identified by the subject query instance (or by its parent object) as a "default" explosion query. In other embodiments, the explosion parametric query may be identified by the subject query instance (or by its parent object) according to a "default" reusable linkage object, as discussed further below. Such a "default"

assignments may allow the user to quickly expand and/or explore a tree data structure. In still other embodiments, the user may provide the explosion parametric query at explosion-time. In one embodiment, the explosion parametric query may be determined in accordance with a user-selected property associated with the subject query instance and/or the query object from which it was instantiated.

[Para 77] In some embodiments, the same explosion parametric query object may be instantiated for each subject result row. In other embodiments, different explosion parametric query objects may be instantiated for some or all of the subject result row.

[Para 78] In block 1215, subroutine 1200 links a result field of the explosion parametric query instance to an input parameter of the explosion parametric query instance. In some embodiments, a "default" linkage may be identified by the subject query instance (or by its parent object). In one embodiment, a linkage may be determined in accordance with a user-selected property associated with the subject query instance and/or the query object from which it was instantiated.

[Para 79] It will be appreciated that although subroutine 1200 illustrates a relatively simple explosion embodiment (for clarity of explanation), in which a single result field is linked to a single input parameter, more complex embodiments may involve a greater number of links and/or input parameters. In such embodiments, subroutine 1200 may be modified accordingly.

[Para 80] In subroutine block 1100, subroutine 1200 obtains one or more result rows for the explosion parametric query instance according to subroutine 1100, as illustrated in Figure 11 and discussed above. In block 1225, subroutine 1200 automatically displays a "child explosion" node according to the explosion parametric query instance, the link established in block 1215, and the result rows obtained in subroutine block 1100. For example, referring to Figure 8, discussed above, subroutine 1200 may display node 842 and linkage 851 on one iteration of block 1225, and may display node 843 and linkage 852 on a subsequent iteration of block 1225.

[Para 81] In ending loop block 1235, subroutine 1200 loops back to block 1205 to process the next subject result row (if any). In block 1240, subroutine 1200 automatically lays out the explosion nodes thus created. In some embodiments, the explosion nodes may be algorithmically laid out so that they do not overlap with existing nodes on a

visualization graph. In other embodiments, some or all result nodes in a visualization graph may be rearranged when an explode operation is performed. In some embodiments, the user may be able to select a positioning configuration for the explosion nodes. For example, the user may select a configuration in which the explosion nodes are arranged vertically, horizontally, surrounding their parent result node, or the like. Additional details regarding automatic node layout may be found in Appendix B to this application.

[Para 82] In some embodiments, a set of explosion nodes may be "unexploded." In other words, some embodiments may provide a facility to automatically delete explosion nodes and/or their associated links.

[Para 83] In various embodiments, via a combination of explode and unexploded operations, a user may be able to essentially jump from node to node, changing queries as desired, to easily traverse a database structure.

[Para 84] An exploded tree structure may provide advantages over a tabular display that ordinarily results from a database query. Exemplary advantages may include some or all of the following:

- The user may be able to select which parts of the tree to expand, reducing the total amount of data displayed and/or highlighting important data.
- The user may be able to identify rows of interest in a complex dataset more quickly and with fewer errors than by crafting a query to explicitly select only rows of interest.
- An exploded tree display may highlight intermediate structures within the data that would otherwise be hidden in a tabular display. For example, a tabular display may hide the fact that many rows are coming from one or a few partial results.
- The linkage mechanism described herein may allow different queries to run on different databases, even if the databases do not support an equivalent cross-database Join.
- An exploded tree display may simplify and clarify logical relationships between pieces of data. By contrast, in a tabular display, intermediate table data values in a complex join are typically repeated many times.

[Para 85] In some embodiments, an explode operation may not create a new exploded result node if an existing node with the same values already exists. Instead, a link may be created to the pre-existing node. The resulting diagram may no longer be a tree, but a general graph. This may further allow the user to recognize a structure in the data that may not be readily apparent in a tabular presentation of the results.

[Para 86] In various embodiments, a data visualization system as disclosed herein may be used in some or all of the following exemplary scenarios:

- *Record lookup.* A data visualization system as disclosed herein may be useful for prototyping, system development, and/or problem solving. For example, a user may be able to investigate related events (e.g., customer events, order events, and the like) even before production screens have been developed.
- *Understanding the data.* A user may be able to easily follow a chain of thought to see various relationships in a database.
- *Constructing charts that present structured data.* Such a chart may show the organization of the database along with data at each level of the database structure. Alternatively, the structure may be within a single table or group of tables. For example, a user may desire a diagram that shows how a series of states occurred, along with the event record that occurred between the states. A user could create a one-off chart with a drawing program, but it would likely require much tedious and error-prone manual data transfer. By contrast, a data visualization system as disclosed herein may reliably and easily create a chart that can be re-used and/or updated with new data at a later time.
- *Comparing the data in different databases.* During system development, a user may wish to compare a complex series of related data items in different databases. Using one or more linkages, a user may be able to quickly generate result nodes showing related items from different databases.
- *Analyzing an application or process.* There are several ways of documenting how an application or process is structured. a data visualization system as disclosed herein allows a user to build a diagram that shows the sequence of operations in an application and the data that is used at each step. For example, a user may enter a

specific customer id at the initial step, for example, and see the rest of the diagram updated to show the data used at that part of the process. Each step can be labeled with a description and label, and the user may quickly navigate to that step be selecting from a list.

[Para 87] Although specific embodiments have been illustrated and described herein, a whole variety of alternate and/or equivalent implementations may be substituted for the specific embodiments shown and described without departing from the scope of the present disclosure. This application is intended to cover any adaptations or variations of the embodiments discussed herein.

VS

APPENDIX A

VS

HIERARCHICAL DIAGRAM SYSTEM AND METHOD

FIELD

[0001] The present disclosure relates to computer-based diagramming tools, and, more particularly, to systems and methods for constructing diagrams of deeply hierarchical data.

BACKGROUND

[0002] Creating a software program that includes the ability to display structured data in a diagram format is currently a time-consuming and complex activity. Many programs are available to draw complex diagrams, and many programs are available for constructing reports from a structured data source such as a database. Programs that do both are uncommon, because it is difficult to do both, and when it is done the resulting program is typically not reusable with other data structures or with other diagram formats.

[0003] Several products are available to help with developing diagram-based tools. For example, the Graphical Editing Framework provided by the Eclipse Foundation, Inc. of Ottawa, Ontario, Canada, is a framework for constructing diagram-based environments. Another such product is JGraph, provided by JGraph, Ltd. Of Northampton, England. Constructing a diagram-based tool with these frameworks requires writing complex software to access the framework application programming interface ("API") to create the displayed objects and send and receive several types of low level notifications. The galleries that are displayed for those products show that what is ordinarily done with those frameworks is constructing interfaces consisting of simple line drawings of nodes and links with limited text and annotations on the nodes. Working at that low level of an abstraction to produce a complex diagram is very time consuming and error prone.

[0004] Existing frameworks may not be suitable for rapidly constructing deeply hierarchical diagrams that can be bound in a simple way to complex data. They may not provide building blocks that can be combined in a graphical environment to build an application that also builds deeply hierarchical diagrams. Existing frameworks may not be suitable for rapidly developing applications containing nested diagrams where the links in the diagrams can cross the hierarchical boundaries, such that the objects in diagrams are mapped to an underlying data source that is not simply a representation of the diagram itself. Existing frameworks may lack the ability to rapidly construct applications that contain several multi-

level diagrams with different structures where changes made to the data objects via one diagram are automatically displayed in the diagrams with the other structures.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] Figure A1 is a diagram illustrating the structure of a software application that has been implemented in accordance with one embodiment.

[0006] Figure A2 is a diagram illustrating the structure of component definitions in accordance with one embodiment.

[0007] Figure A3 is a diagram illustrating the expansion of a component definition in accordance with one embodiment.

[0008] Figure A4 is a diagram illustrating various classes used to build component definition contents in accordance with one embodiment.

[0009] Figure A5 is a diagram illustrating a software facility to assemble components in accordance with one embodiment.

[0010] Figure A6 is a diagram illustrating structures used to perform change notification in accordance with one embodiment.

[0011] Figure A7 is a diagram illustrating a software workbench for assembling applications in accordance with one embodiment.

[0012] Figure A8 is a flow diagram illustrating an exemplary hierarchical diagram creation routine in accordance with one embodiment.

[0013] Figure A9 illustrates a hierarchical diagram, such as may be created using one embodiment.

DESCRIPTION

[0014] The detailed description that follows is represented largely in terms of processes and symbolic representations of operations by conventional computer components, including a processor, memory storage devices for the processor, connected display devices and input devices. Furthermore, these processes and operations may utilize conventional computer components in a heterogeneous distributed computing environment, including remote file Servers, computer Servers and memory storage devices. Each of these conventional distributed computing components is accessible by the processor via a communication network.

[0015] The phrases "in one embodiment," "in various embodiments," "in some embodiments," and the like are used repeatedly. Such phrases do not necessarily refer to the same embodiment. The terms "comprising," "having," and "including" are synonymous, unless the context dictates otherwise.

[0016] Reference is now made in detail to the description of the embodiments as illustrated in the drawings. While embodiments are described in connection with the drawings and related descriptions, there is no intent to limit the scope to the embodiments disclosed herein. On the contrary, the intent is to cover all alternatives, modifications and equivalents. In alternate embodiments, additional devices, or combinations of illustrated devices, may be added to, or combined, without limiting the scope to the embodiments disclosed herein.

[0017] Various embodiments are described of a system that allows a user to develop software applications that may contain hierarchical diagram tools, where the diagrams may be bound to structured data from external sources. The hierarchical diagrams produced by the applications resulting from this system allow users to show connections between data objects visually, even when the relationships are between data objects at different levels in the structure. The applications may be built using a library of component building blocks that handle data binding and change propagation. The user can develop new components and so, for example, build a library of domain-specific diagrammatic components.

[0018] An embodiment of the system shown in Figure A1 allows the user to construct a software application by creating and assembling component definitions A120. The assembly facility A125 may include a library of component definitions A120 and may provide the capability for users to create new component definitions A120. Using an embodiment of the system, a user may connect those component definitions A120 together by making references to other component definitions A120. An embodiment of a target application that is built using the system may run by creating an initial component instance A105 that recursively creates additional component instances A105 according to the component definitions A120 through a process called selective expansion A115.

[0019] Components instances A105 may interact with external data and events A110 and may control internal objects A100. The external objects A110 may be instances of classes that are provided as part of a graphical system that makes the application visible to an end user. Internal objects A100 allow data to be transformed before it is rendered in the output. The selective expansion A115 process maps the structure of component definitions A120 into a

structure of component instances A105 in accordance with the state of internal objects A110. When objects in the system change their state, change notifiers A135 notify other objects throughout the system so that they may update their state accordingly.

[0020] In one embodiment, each object may be a region of computer memory and a set of functions that are associated with the class of object. Other objects, even within the same embodiment, may have one region of memory for some data values and additional regions of memory for other values.

[0021] A class may inherit from another class, which means that the derived class may provide different implementations of functions of the same name that appear on the class it inherits from (which is called the base class) and/or additional functions. Objects of the derived class may use additional amounts of memory. When a function takes an argument that is a member of base class, a function on a base class also works with a member of a derived class, unless noted otherwise.

[0022] In these figures, each square box represents a class of objects or a group of related classes. The explanations of the figures often refer to representative objects by their class names. The elements shown in Figure A1 are part of the underlying structure of the system and are implicitly present in all the figures. Reference will be made back to Figure A1 for these elements.

[0023] In the following figures, the notation "0..*" on arrows indicates zero or more occurrences of the class of object shown at the head of the arrow can occur for each occurrence of the class of object shown at the tail end of the arrow. An embodiment may implement the arrows shown in these figures using a data structure that supports a number of features, including the ability to follow references in both the forward and reverse direction and the ability provide a list of all the objects that reference a given object in the context of a given relationship. The arrows on the figure are meant to illustrate the major relationships. An embodiment may need additional relationships described in the narrative.

[0024] An embodiment may use the data structures that implement the arrows to notify affected objects when the state of various objects has changed, such as objects being added or removed from a set. Because each object may be referenced in many relationships, an embodiment may maintain a tag value on objects that indicates for example, when an object is being deleted, so that the deletion code is not executed more than once on the same object.

[0025] COMPONENT DEFINITIONS

VS

[0026] To explain how component definitions A120 work, it may be useful to consider how a function call works. Many programming languages share a fundamental pattern wherein a function with parameters is defined and expressions are created which call that function and which provide argument expressions. At run time, argument expressions are evaluated and the resulting values are passed to the corresponding function parameters. When the function call is completed, then control returns to the calling function, and resources associated with the function call are released.

[0027] An embodiment of Figure A2 may have objects that are similar to expressions and parameters and structure, but unlike a function call, these objects remain active as long as the component instance (not shown) that invokes the component expression A210 is valid. In particular, when the value of an expression on a component argument A230 changes, the embodiment may pass the updated value to the corresponding component parameter A225 even after the call has been activated. Figure A2 shows the static relationship of objects that make up a component expression A210. Figure A3, described below, will show the objects used in an invocation.

[0028] In one embodiment, objects such as component definitions A120 A and B and related objects may be created using an ordinary text editor. (See Figure A5, discussed below). Alternative embodiments may provide a graphical environment for creating those objects. (See Figure A7, discussed below).

[0029] In Figure A2, there are two component definitions A120A and A120B. These are analogous to how, in a conventional programming language, one function can call another function. A component expression A210 is the object that provides the argument expressions whose value is passed in an invocation of a component definition A120 B or component type A200. In an embodiment, such an invocation results in the creation of the component instances A105 (see Figure A1).

[0030] A user that develops a component definition A120B may define component parameters A225 that describe the parameter values that need to be provided when a component expression A210 refers to that component definition A120B. Component types A200 also have component parameters A225, but in that case the parameters correspond to the public data members and/or setter functions that may called on external objects and events A110 (see Figure A1). The difference between component types A200 and component definitions A120B is that component types A200 describe object classes that are defined

outside this system, whereas component definitions A120A are defined using an embodiment of this system. A component instance A105 created for a component type A200 may have a pointer to a conventionally implemented object, whereas a component instance A105 created for a component definition A120A will not.

[0031] Each particular component definition A120A may create a list of zero or more component expressions A210. An embodiment may create this list by evaluating an expression that is stored on the component definition A120A. Techniques for the creation of a component expression list as indicated by arrow A205 will be described later in relation to arrows A270 and A285.

[0032] In this discussion, the term component expression is used because it is an extension of the concept of a function call expression in a conventional programming language. There many other places in this system where the term expression is used. Those expressions refer to some embodiment of the conventional concept of a section of source code that is executable and returns a value. As in certain programming languages such as Lisp, the concept of an expression includes nested control structures. Many expressions may contain references to definitions made elsewhere. An embodiment may provide a searchable collection for those definitions so that a name can be used as an expression, or an element of one, to refer to a named definition. An embodiment may place definitions in multiple files, and use compound names to refer to a location and a definition in that location.

[0033] At this level of abstraction, the programmer can rely on the embodiment of change notifiers A135 (see Figure A1) to ensure that the component argument A230 expressions are re-evaluated as needed when a change occurs in the state of the internal objects A100 (see Figure A1) they depend on. The resulting values are forwarded by change notifiers A135 to provide updated values to other objects that originally relied on the out of date values. In particular, an updated value provided to a component parameter A225 for a component type A200 may to be applied by the embodiment to the corresponding external object associated with the component instance A105.

[0034] The component types A200 describe the interface used by an embodiment to access external objects and events A110. An embodiment may obtain the details of the interface to those objects by parsing the source code or other computer generated data for each class of external object and storing the interface information by creating component type A200 and component parameter A225 objects.

[0035] The response A235 objects hold the programming statements that are executed in when an event is received from external objects and events A100. The response action is typically is to update one of the properties of the internal objects A100 that were passed to the component instance A105 when the component instance A105 was created. As illustrated in Figure A3, that update will propagate back through all of the affected component instances A105 and through them the external objects and events A110 will reflect the new state of the internal objects A100.

[0036] The purpose of component expressions is to create objects. Binding expressions A240 are oriented towards making connections between existing objects. Each binding expression A240 may have a pair of expressions that refer to objects that need to be connected via a relationship when a component instance A105 is created. Those objects may also be disconnected when the component instance A105 is deleted. A binding expression A240 may hold a reference to a binding definition A250 that may hold the references to the functions to connect and disconnect. The object expressions on a binding expression A240 may be evaluated by an embodiment to direct references to the objects being controlled, or they may evaluate to be references to the component instances A105 from which the controlled objects are obtained. An embodiment may provide that if one of the pair is missing that it is an implicit reference to the component instance A105 that is being expanded at that time.

[0037] A programmer may use component expressions A210 for maintaining many kinds of relationships. For example, the objects in a graphical display are typically arranged in a tree data structure and so in those cases the component definitions A120 need to make calls on the graphic library (represented here by external objects and events A110) to maintain the required tree structure by adding and removing controlled objects at the necessary location in that tree. In addition, the component instances A105 themselves form a network and so when a component instance A105 is expanded to create lower level component instances A105, the embodiment may be add those lower level component instances A105 to a network data structure.

[0038] In addition to the tree structure and component instance A105 networks described above, an application typically requires that additional logical relationships also be implemented. To keep the component definitions A120 tractable, an embodiment may provide additional data structures to assist in the creation of the list of component expressions A210 indicated by arrow A205. One approach to this is represented by view entry A280. An

embodiment of a view entry A280 may be a component definition A120. The set of component expressions A295 that a view entry A280 creates, shown by arrow A285, may be determined by invoking a function that is defined on a view item A265 and returned to the view entry as shown on arrow A270.

[0039] In order to improve the reusability of view items A265, an embodiment may use arrangements A260 to provide specific values that are provided to a shared view item A265. Details on how an embodiment can use view items A265 and arrangements A260 are shown below in Figure A4.

[0040] INSTANTIATION

[0041] Figure A3 shows how the component definitions may be recursively expanded. Parent component instance A105A represents a component instance A105 that has been created for a particular component definition A120. The set of component expressions A210 created for a component definition A120 shown as arrow A205 is intended by the abstraction to be mapped into a set of components instances A105B. An embodiment may use a mapping A305 object to control that expansion.

[0042] An embodiment of mapping A305 may use the structure shown in Figure A6, described below, so that if the internal objects A100 (see Figure A1) that are accessed during the expansion are later changed, the set of component instances A105B that resulted from the expansion may be enlarged or reduced, and/or its members updated. Such a change could occur because an application changed the state of the internal objects A100 or because the user made an edit to the component definition A120.

[0043] A component parameter instance A315 is an object that represents an occurrence of a component parameter A225 (see Figure A2). Referring back to Figure A2, there are two cases to consider, depending on whether the component instance A105B resulted from a component type A200 or a component definition A120B. If the component parameter instance A315 is associated with a component type A200, then, when a value is obtained from the component argument A230, that value may be applied to an external object. To do that, an embodiment may use the setter function stored on the corresponding component parameter A225. If the component parameter instance A315 is associated with a component definition A120B, then when a value is obtained from the expression on the component argument A230, that value may be accessed by the expressions that occur in the expansion of the component definition A120B. An embodiment may make those values accessible by implementing a

symbol table associated with each component instance A105, such that when a symbol reference is made from within a component argument A230 expression, for example, using the name of a component parameter instance A315.

[0044] Referring again to Figure A3, the response objects A235 are where the user puts the code that is to be executed when an event is send to a component instance A105. An embodiment may create responder A320 objects and register them with the external objects and events A110 (see Figure A1). When an event occurs, the external objects and events A110 system may call a function on a listener object that is created and registered by an embodiment of responder A320. That listener may pass control back to the responder A320, which may execute the code that is associated with the corresponding response A235. An embodiment may use automatic code generation to create the code for the listener object that is actually registered. That way the code can conform to the interface needed by the external objects and events A100 without burdening the user with the need to write that low level code.

[0045] Recall that a binding expression may be used to connect objects together. An embodiment may create binding instances A325 to implement occurrences of the binding expression A240 objects that correspond to a particular component instance A105.

[0046] An embodiment of a component instance A105 may use instances of the mapping A305 to maintain the sets of component parameter instance A315, responder A320, and binding instance A325 in accordance with their corresponding expression objects.

[0047] To simplify locating component instances A105, an embodiment may register them in register tables A310 that are indexed by some type of id assigned to each component instance A105. For example, an embodiment may use an automatically generated unique name to identify them. This allows a reference to a component instance A105 to be made even before that object is created, provided the component instance A105 is eventually registered under that key by the time the reference needs to be resolved. An embodiment may use a similar table to register component instances A105 under a key that is, for example, three objects that can be called a scope, a tag, and a basis. In this example a scope may refer to some kind of diagram that is being constructed, as identified perhaps by a particular arrangement A260 used to implement that diagram. A tag would optionally refer to some value to indicate how the particular component instance A105 is to be used, such as for example, for an input versus an output connection point. The basis may be a reference to some object for which that

occurrence of a diagram is being created. Later when, for example, a user is using an application to draw a link to the input or output connection point, an embodiment would have the information it needs to locate correct component instance A105 using the register table A310.

[0048] SELECTIVE EXPANSION

[0049] Figure A2 introduced the view item A265 to reduce the complexity of component definitions A120A. Figure A4 shows the relationship among several classes derived from view item A265 that an embodiment may provide. These different subclasses will each use additional parameters. To help specific view items A265 be more reusable, an embodiment may store some or all those parameter values on arrangement A260 objects.

[0050] As each of these objects is expanded first into component expressions A210 (see Figure A2) and those into component instances A105B (see Figure A3), there may be some particular internal object A100 (see fig 1) where displayed data will come from and on whose state run-time decisions are based on. That object is called the "basis." The various parts of an application may have a different basis, and the basis of each part may change as the result of user input or other events. To allow the expansion to not have to be repeated every time a different basis is selected, an embodiment may avoid references to the basis except on certain objects.

[0051] Referring now to Figure A4, an embodiment may keep a reference to the basis on the controller A400 instead of being on arrangements A260 or view items A265. If a new component instance A105 (see Figure A1) needs to refer to a different basis, then during expansion an embodiment may construct a new controller A400 to refer to that other basis. These controllers A400 may be organized into a tree, and when needed, an embodiment may traverse up the tree to find needed information that is stored at a higher level.

[0052] An embodiment may initiate the process of expansion of a view item A265 by constructing a view entry A280 and passing to it a controller A400 that may hold a reference to an arrangement A260 or view item A265. An embodiment may also pass a view item A265 to a view entry A280 to override the view item A265 that would otherwise be obtained from the controller A400 and yet otherwise use the controller A400 for its position in the tree of controllers A400.

[0053] An embodiment may inform a view entry A280 where generated component instances A105 are to appear in various structures. For example, the component instances

A105 that control graphic objects may need to obtain a reference the component instance A105 that controls the parent of the graphic object. The embodiment may also organize the component instances A105 into a hierarchy in order to support memory management functions. In both of those cases, an embodiment may pass to the view entry A280 a unique identifier to identify a particular component instance A105 that has not been created yet. The actual component instance A105 can be located later by looking up the unique identifier, provided that the embodiment registers each new component instance A105 with its unique identifier(s).

[0054] The following paragraphs describe various specialized versions of the view item A265 that an embodiment may provide. In the following discussion, one or more arrangements A260 may be mentioned, but an embodiment may find a view item A265 a more convenient object to use in some cases.

[0055] A view tree A405 has view tree contents A407, which are view items A265 or arrangements A260, and optionally a manager A406, which may be an arrangement A260. An embodiment of a view tree A405 may construct a list of view entries A280 for the view tree contents A407, and if needed, for the manager A406. The embodiment may generate unique placement identifier for the manager A406 which it passes to the view entries A280 generated from the view tree contents A407. This is so that the component instances A105 that are generated for view tree contents A407 are able to use the placement identifier to locate the component instance A105 that is generated for the manager A406.

[0056] In some cases, the manager A406 may result in the creation of multiple component instances A105. An embodiment may control which one of those gets the placement identifier as follows. Referring to Figure A3, an embodiment may pass the placement identifier to the functions that perform the expansion A205 so that the component instance A105B that is assigned to get that placement identifier is the one where child component instances A105B should connect their controlled objects to as their parent. Component definitions A120 that generate more than one component expression A210 (see Figure A2) may pass the placement identifier to the particular component expression that is to control the object that is to be the container for the objects controlled by the component instances A105B generated from the view tree contents A407 (referring back to Figure A4).

[0057] A view conditional A410 takes a list of clauses, one of which is selected to be expanded. A clause may hold a reference to an expression that is evaluated to determine if that

is the clause to be expanded. Once selected, the contents of the clause may be expanded in a manner similar to a view tree A405.

[0058] A view single A415 may be used to build a single component expression A210 (see Fig. 2). To reduce repetition, an embodiment may gather commonly occurring component arguments A230 (Figure A2) that are to appear in the resulting component expression A210. One way that an embodiment may do that is to put executable code for producing the component arguments A230 on behavior definitions A425. To use the behavior definitions A425, the embodiment make defined one or more behavior instances A420 to reference that behavior definition A425 and also may identify the type of object that the component instance A105 (Fig. 1) should create by referencing a component type A200 or a component definition A120.

[0059] A view reference A440 provides a way to shift the expansion to a different basis object and/ or a different arrangement A260. An embodiment may obtain expressions from the view reference A440 and evaluate them to return the new basis object or arrangement A260. An embodiment of a view reference A440 may provide additional arrangements A260 to be selected for expansion in special conditions such as if the expression does not return a basis object or if an error is encountered. A controller reference A430 is a class derived from controller A400. The purpose of a controller reference A430 is to choose which arrangement A260 to be expand depending on whether the basis expression returns a value, null, or an error is encountered.

[0060] The view mapping A445 provides a way to expand an arrangement A260 once for each member of a base list. The base list may be obtained by accessing a list type attribute value from the basis object, where the attribute is identified by an expression that is obtained from the view mapping A445. An embodiment may use a mapping A205 (see Figure A2) and change notifiers A135 (see Figure A1) so that when objects are added or removed from the base list value then the embodiment will expand the arrangement A260 on the view mapping A445 or remove the previous expansion, respectively. If a change occurs to one or more properties of an object that is a member of the base list, then the mapping A205 may isolate the change to the corresponding member of the expanded arrangements and so avoid having to re-expand the arrangement A260 for the other objects in the base list. An embodiment may implement that isolation by creating a transaction A610 (described below in Figure A6) for

each element of the base list and a dependency A605 (see Figure A6) that refers to the corresponding element of the base list.

[0061] The view grid A450 provides another way to expand the elements in the list. The difference from a view mapping A445 is that instead of expanding a single arrangement A260 once for each element of the base list, a view grid A450 has list of arrangements A260 that an embodiment may expand for each item. An embodiment may implement the expansion of a view grid A450 by using two component definitions A120. The first uses a mapping A205 for the base list value and a second mapping A120 for the arrangement A260 list. The first mapping expands into a list of component expressions A210 that refer to the second component definition A120. The embodiment may expand the second component definition A120 by expanding the list of arrangements A260 on the view grid A450 in the same way as was described for a view tree A405. The embodiment may expand the view grid A450 into component instances A105 that result in the creation of a flat list of graph objects and rely on the graphics library to arrange the resulting object instances into the desired format such as a grid. An embodiment may use a binding expression A240 (see Figure A2) to create an object to hold the parameters to the graphics library object that controls the layout. The binding definition A250 (see Figure A2) may hold references to the functions that need to be called to attach the layout object to the graphics library object that contains the grid objects.

[0062] An embodiment of a view grid A450 may need a way to connect the graphic container object and the layout manager required by the graphics system. An embodiment may walk up the tree of controllers A400 to find the controller A400 that is tagged as being the controller where the view grid A450 is expanded. Since the embodiment of each branch of the expansion is able to access to the same controller A400, those component instances A105 are able to place information on that controller and in this way the embodiment is able to connect the container graphic, the layout manager, and the child graphic objects.

[0063] The view column A455 extends the concept of the view grid A450 by using a list of descriptors A460 instead of a list of arrangements A260. A descriptor A460 is an object that an embodiment may use to hold values such as heading text and an attribute value. The embodiment may provide a second level component definition A120 (see Figure A1), in a similar manner to how the view grid A450 is implemented. The second level component definition A120 for the view column A455 may construct a list of arrangements A260 using

the information from the list of descriptors A450. At that point, the view column A455 may follow the description given for the view grid A450.

[0064] A possible usage of the view grid A450 and view column A455 is to construct a table. An embodiment of a table may create a pair of view grid A450 or view column A455 with one resulting in the creation of a row of graphic objects for table headings and the other a series of rows of detail graphic objects, with one row for each object in the base list. The embodiment may place the pair of view grids A450 or view columns A455 in the view tree contents A407 of the same view tree A405, and since both the header and detail items share the same manager A406, they will appear in the same graphic container object. An embodiment may use a standard grid layout manager to align the corresponding heading and detail objects.

[0065] The view formation A465 provides the ability to use a list of descriptors A460 such as used for a view column A455 but applying it to the basis directly instead of to a list attribute of the basis. An embodiment may implement this by providing a component definition A120 that is similar to that used for the second level of a view column A455.

[0066] The arrangements A260 that an embodiment constructs for each descriptor A460 in a list that is referenced by a view column A455 or view formation A465 may be embedded in a wrapper arrangement A260. This may be used for example to display a label to identify each field. An embodiment may implement the creation of the wrapper arrangement A260 with a function that is called for each descriptor A460. That function would create a copy of a base arrangement A260 for the label and construct a view tree A405 to enclose both the label arrangement A260 and the arrangement A260 that is constructed using the information from the descriptor A460.

[0067] The view selector A470 is an alternative to the view conditional A410 that is specialized for selecting a clause based on some property of the basis. In one embodiment, a selector definition is created and is referenced by the view selector A470. Then the embodiment may define clauses that, when they are constructed, call a function to register with the selector definition along with an expression for the condition. The expansion for the view selector A470 may then proceed for each registered clause in the same manner as for a view conditional A410.

[0068] ASSEMBLY FACILITY

[0069] Figure A5 shows details of the assembly facility A125 (see Figure A1) and shows a structure that may be used for internal objects A100 (see Figure A1). Internal objects A100

may be broken into smaller parts so that change notifiers A135 can operate on those parts in such a way that a change to one part does not require redoing calculations that depend only on different parts.

[0070] The model A500 represents an interface that describes objects that have a number of attributes, where each attribute has a name and value, and a number of functions. An embodiment may implement this interface on several types of classes so that features that expression evaluation can work all of the implementation classes in the same way.

[0071] An embodiment of a model A500 may use the data definition features of the implementation programming language to take advantage off the shelf support for compiler type checking, syntax directed editing, and debugging. An embodiment may also use an internally defined data structure in order to perhaps simplify how a user is able to create new model definitions at run time.

[0072] An embodiment may use a model definition A505 as place to store functions that are needed to access an implementation of the model A500 interface. The embodiment of a model definition A505 may extend another model definition A505 in order to provide inheritance as described in standard object-oriented programming.

[0073] An embodiment of a part definition A515 provides a place to store details about data members associated with a model definition A505, such as name, type, default value expression, identification used in external files, whether the value of the data member is to be written to files, and the like. The part definition A515 may identify functions to call when a data member described by that part definition A515 needs to be created, deleted, or modified. The part definition A515 may also have an inverse, which is a different part definition A515 that may belong to a different model definition A505. Of the two part definitions A515 that are in an inverse relationship, one is single valued and the other is list valued. The list may be implemented using the inverse A635 (see Figure A6) chain.

[0074] The part base A510 represents an interface that an embodiment may use to provide access to objects that implement value-holding objects that are described by a part definition A515. A single embodiment may have multiple implementations of the part base A510 interface. An embodiment may mix the implementations as needed, for example to use native programming language features for some parts and run-time definable parts for others. An embodiment may also provide a run-time modifiable implementation of a model A500 and/or part bases A510 for development of an application and then use a code generator to produce a

functionally equivalent native code implementation that is used in production use of the application.

[0075] For a native code based implementation of data objects that are described by model definitions A505 and part definitions A515, an embodiment may use the reflection facility of the native environment if it provides one. For capturing additional information that cannot be obtained from reflection, the embodiment may use an annotation capability if that is available. In cases where neither is available, an embodiment may import A530 the model definitions A505 and part definitions A515, as described below.

[0076] An embodiment may use the symbolic A520 class as a base class for many its objects that have multiple data members. An embodiment of the symbolic A520 class does not have to implement the model interface A500 directly. Instead, several classes that extend symbolic A520 may implement model A500 in different ways. Nevertheless, the model definition A505 and part definition A515 may be used to provide uniform access to instances to instances of symbolic A520.

[0077] Examples of classes derived from symbolic A520 that an embodiment may provide include the standard programming language constructs such as functions, local variables, assignment, conditional, iteration, sequencing, and function calls. An embodiment may provide data types and operators such as arithmetic, string and list operators. An embodiment may also provide the model definition A505 and part definition A515 objects to provide access to other classes.

[0078] An embodiment of a symbolic A520 may use part instances A525 to hold the run-time values for data members of an instance of symbolic A520. The native programming environment may be used to hold the values, but change notifiers A135 may require additional information for each part instance A525 and that information may be stored on a separate object for each part instance A525. One way to combine the benefits provided by an off the shelf programming environment with part instances A525 is to have the embodiment classes hold references to the part instances A525 as data members. For example, if a data member named m_size refers to a part instance A525, then the value may be accessed in the Java programming language with code like m_size.get().

[0079] Each part instance A525 may hold a reference to the part definition A515 that holds references to functions that describe the behavior for that part. One such function in particular is called the value method A526. If an embodiment requests a part value associated with a

symbolic A520, and no part instance A525 exists with a valid value, the embodiment may invoke the value method A526. Change notifiers A135 (see Figure A1) may later update that value as shown in Figure A6. Besides a value method A526, an embodiment may hold a reference to a function that is to be invoked when a part instance A525 is deleted. Such a delete method may be used, for example, to delete the link objects connected to a node object when that node object is deleted.

[0080] An embodiment may provide a file format and a class to export A525 object instances to that file. An embodiment may also provide a corresponding import A530 class. An embodiment may use model definition A505 and part definition A515 objects to serialize and de-serialize various types of object so that the embodiment can store objects in external files A545. The objects that are imported A530 or exported A535 are not necessarily only those described by symbolic A520 in terms of their purpose in the embodiment.

[0081] One possible embodiment of the format of external files A545 is as follows.

[0082] The file format used may be human readable and editable so text oriented merge tools may be used resolve conflicting updates. Such a file format may use natural language names to allow the files to be a suitable environment to develop applications in using a text editor. The use of names helps to minimize the multi-user problem of id numbers getting out of sync in two different file versions. The names may also be aliases to help with code obfuscation. An embodiment may also use a file format with a compact format that is not intended to be read by a user.

[0083] Because the data may be hierarchical, the file format may use indentation, bracket characters, or some other way to represent nesting. An alternative is to use numeric or other references to indicate the parent object to which imported objects are to be applied to as the file is read. The following examples illustrate an embodiment that uses indentation.

[0084] Each line in the external file A545 may set an attribute or create an object, and in some cases, one line does both. As an embodiment of import A530 reads an external file A545, it constructs a tree of objects. Each time the import A530 object reads an identifier that refers a model definition A505 it calls a function from the model definition A505 to construct the corresponding type of object. The identifier may be the implementation language name of a class or an alias. The identifier of the model definition A505 is indented to be at least one character position in from its parent object. Identifiers that appear at the outermost level of indentation means that the object is to be added to an object that represents the contents of

the external file A545. A method that the embodiment may use to find the function to add the objects to its parent is described below.

[0085] A part instance A525 is represented in an external file A545 as a line of the form "part-identifier=part-value". The function to set the value on the part is obtained from the part definition A515 that matches the part-identifier, within the scope of the model definition A505 if there are conflicting identifiers. The part lines are indented underneath the line for object that the part-identifier applies to. For each model definition A505, one of its part definitions A515 may be defined as primary. For the primary part, an embodiment may read the value immediately after the model definition A505 identifier. The function used to convert from the format used externally to the internal representation may be obtained from the part definition A515.

[0086] Here is an example of a few lines from a file that would work with such an embodiment.

```
ClassA value1
   Attr2=value2
   ClassB
     Attr3=value3
```

[0087] That would cause the creation of two objects and set three part instance A525 values. The value1 is assigned to the primary part instance A525 of the ClassA object. The relationship between the ClassA object and the ClassB object will be described below.

[0088] Some parts are defined with a primitive type, such as string or literal. In other cases, the value is one or more objects. In that case, the part is identified with a name called a role. Again, one of the roles can be primary in addition to the primary primitive type part. A non-primary role is named and followed by a colon, as in this example.

```
ClassA value1
   Attr2=value2
   child:
     ClassB
       Attr3=value3
```

[0089] That is the same as the previous example, but the role used to add the ClassB object to the ClassA object is given explicitly.

[0090] An embodiment of a part definition A525 may allow the role value to be set using an expression, which when evaluated returns a reference to an object. When the object is later exported, it either uses the original expression or constructs a new expression. An embodiment of a part definition A525 may indicate that the evaluation is to be done using

symbols that are visible statically. Such a value can be cached. In other embodiments of part definition A525, the evaluation may be done in a context that allows run-time values, which means that those expression values may not be cached for use in other contexts.

[0091] When a part definition A515 allows a part value to be an expression, the embodiment may provide a class that takes as its value a string that contains an expression written in another syntax, such as infix notation, which means the operators come between the operands as in, for example, a+b. When a function is called on that object which requires its value, the object parses the string value and then evaluates the parsed result. An embodiment can cache the parsed value or convert it to some other executable format.

[0092] The functions that may be stored in the model definition A505 to create instances of objects may perform other actions as well. In some cases they may cause the object just loaded to be evaluated immediately or after the file it appears in has been completely read.

[0093] One technique for writing a set of related definitions that may be used is to define macros. A macro is a function that constructs a data structure that represents executable code and then executes that code to produce its result. The data structure for a macro may use a technique called a backquote. A backquote expression looks like an ordinary expression except that there are places where additional executable code is placed. Those places are identified in some programming languages by a comma. When the backquote expression is evaluated, it creates a copy of the expression and as the copy is made, the comma expressions are evaluated and the results for each substituted into the copied backquote expression at the location of that comma. This standard concept of a macro can be applied in an embodiment by recognizing comma expressions where attribute values and role objects appear. An embodiment may save those expressions on the backquote data structure and evaluate them when a copy is made. The resulting value is applied to the copy using the same routines that are used for the expression or role where the comma expression was located.

[0094] An embodiment may provide an export A535 corresponding to the import A530 behavior described above. An embodiment of export A535 may traverse a tree of symbolic A520 objects that are to be exported. An embodiment may perform the traversal using the list of part definitions A515 associated with the model definition A505 for the object being exported. The conversion of a part value to an external format can use the native conversion provided by the programming environment or by getting a conversion function from the part definition A515. An embodiment may keep track of the indentation level needed for each

object and insert spaces in the output file, or the embodiment may use brackets or the techniques mentioned above in regards to import to represent the exported tree structure.

[0095] The external file A545 may be used to hold an application or application data, or both.

[0096] An application may be a collection of component definitions A120 together with related view item A265, arrangement A260 and other objects. A user may launch such an application at the operating system level by invoking an embodiment of a base program which runs the import A530 operation on an external file A545 that is identified on the command line or other location. If that external file A545 contains a reference to an object whose model definition A505 causes it to be evaluated during import A530, then the effect will appear to the user as if a program is running that is defined by the contents of that external file A545. Using a compatible file format for both data and program allows the same embodiment to both develop applications and to run the resulting application. Alternatively, an embodiment may use code generation to convert the symbolic A520 objects and their part instances A525 into equivalent executable code in another programming language or directly into machine executable code.

[0097] External files A545 using a file format as described, or similar to that, can be manipulated with an off the shelf text editor A550. Alternatively, an embodiment may provide a workbench A540 environment for the user. A user of the embodiment of a workbench A540 loads an application by performing a user interface operation that invokes the import A530 operation and saves an application using the export A535 operation. An embodiment of such an application may also use the import A530 and export A535 functions to save the data and other objects constructed by those applications. More details on the workbench A540 are shown in Figure A7, which is described below.

[0098] CHANGE NOTIFICATION

[0099] Figure A6 illustrates the change notifiers A135 (see Figure A1) that an embodiment may use to provide change notification. There are two approaches shown. The first uses dependencies A605 and transactions A610. The second uses constraints A655, anchors A660, and observers A665.

[00100] The concept behind dependencies A605 and transactions A610 is that whenever a core A600 is created, a reference to it is added to the current transaction A610. During the execution of a value method A526, when an access is made to a core A600, a dependency A605

is created that references that core A600. The dependency may be added to a list of dependencies on the current transaction A610.

[00101] When a change occurs, for example a core A600 is deleted, an embodiment may call a notification function on all the dependencies A605 that refer to the changed core A600. An embodiment may create a change object that has information about the change that occurred and pass the change object to the notification function. The notification function for a dependency may call a notification function on the transaction A610 that contains the dependency A605. An embodiment of the transaction A610 notification function may call a core A600 notification function on each of the core A600 objects that were created in that transaction. The change object is passed to each notification function in this cascade of notifications.

[00102] An embodiment of the notification function for core A600 may examine the change object to see if it can handle the change. If the core A600 determines that it cannot handle that change, then it returns from the notification function with an indication that the transaction A610 needs to be deleted. If a transaction is so notified, then it may delete all the core A600 objects in its list.

[00103] When a change is made to the value of a part instance A525, one way for the embodiment of the notification method to determine if it can handle a change is to store a notification method on the part definition A515. A part instance A525 notification method may examine the change object and update the existing part instance A525 value if that is possible, or return a failure code if it cannot. When an existing part instance A525 value is modified, the embodiment may call the notification code on all the dependencies A605 in that case so that objects that depend on that value can be updated.

[00104] The value method A526, as described above, may be executed when there is a request for the value of a part instance A525 on a particular symbolic A520 that does not have a valid value. When a value method A526 is executed, it may create a part instance A525 and store a value on it. A transaction A610 may be created at the start of the execution of the value method to gather the dependencies related to that new part instance A525. The embodiment may place the newly created part instance A525 on the list of objects owned by the transaction created for the execution of the value method A526. As a result, when one of the objects referenced by the dependencies A605 in that transaction is changed, the embodiment may call the notification method for that dependency A605 as described above, and that may cause in

the part instance A525 to be deleted. Consequently, the next time the value for that part is requested, the value method A526 may be called again to compute a value using the updated state of the other objects. Value methods A526 may create other core A600 objects besides the requested part instance A525 whose value was requested. If during the execution of a value method A526 another transaction needs to be created, the embodiment may save a reference the previous transaction as a local variable and restore it to being the active transaction at the end of the nested value method A526.

[00105] Part instances A525 and other core objects A600 may also be created during the response to external objects and events A110. An embodiment may create a transaction A610 at the start of the handling of each such event. For purposes of providing an undo capability, an embodiment may place in that transaction A610 all the core A600 objects that are created or deleted during that event handling, except those created during execution of a value method A526.

[00106] An embodiment may provide an undo capability by keeping a list of transactions A610 created for handling input events and then a user interface operation that calls an undo function on the most recent of those transactions A610. When a transaction is undone, the embodiment may delete each of the core A600 objects created in that transaction and those objects that were deleted in the transaction are restored to a valid state. Core A600 objects that were created in a value method A526 do not need to be saved for restoration because they can be recreated as needed by executing the value method A526 again.

[00107] The symbolic A520 described earlier inherits from referent A615, which an embodiment may use for an object that needs to hold a list of pointers to the objects that refer to it. Such a list may be used when an object is deleted to remove all the references to it. A referent A615 may also have a pointer to an owner object, such that when an object is deleted then any referent A615 that has that object as its owner is also deleted. Referent in turn inherits from core A600, the purpose of which is to allow other objects to be notified when that object is modified.

[00108] The embodiment of part instance A525 objects may inherit from referent A615. The deletion of a part instance A525 from a symbolic A520 leaves a gap. Those gaps can be used to drive the update of a graphical display as follows.

[00109] The embodiment of component instances A105 may inherit from symbolic A520. When an event occurs, after the response A235 (not shown) has been executed, the event

handling code may request that affected component instances A105 be updated. An embodiment may use a part instance A525 to indicate if each component instance A105 is up to date. If a part instance A525 is missing, perhaps because it was deleted as a result of a notification function, or because the component instance A105 was just created, then that component instance A105 needs to be updated. The update code may be located in a part value method A526 for the component instance A105, so that as that function is executed, it will create dependency A605 objects as described above, and during the update function it will create a part instance A525 to indicate that component instance A105 is up to date.

[00110] Referring to Figure A3, one of the operations in the update function for a component instance A105A is to construct the list of component expressions A210 according to the component definition A120, shown on Figure A3 by arrow A205. That construction is done in a transaction associated with the component instance A105A so that if any internal object A100 (see Figure A1) is accessed, a dependency A605 is created. When an accessed object is later modified, the dependency A605 attached to it may later notify the component instance A105A that the list of component expressions needs to be rebuilt.

[00111] The update function for a component instance A105A may invoke as needed the update function on subordinate component instances A105B. It may also call analogous update functions on its component parameter instance A305 and binding instance A325 objects, which also use a similar update part instance A525. In that way, an embodiment may request the update part for only the root level component instances A105A and all the lower level objects may be updated indirectly.

[00112] The part value method A526 (not shown) for the update function on the component parameter instance A305 may evaluate the component argument A230 expression, which may result in side effects such as updating the external object and events A110 (not shown). The binding instance A325 objects work in a similar fashion, with the optional addition that the binding definition A250 object may provide a delete function to be called when the binding instance A325 is deleted.

[00113] Returning to Figure A6, an embodiment may provide a chain A620 base class and several derived classes so that different kinds of list behavior can be used throughout the embodiment. An embodiment of chain A620 has the usual member functions for a list such as insert, remove, iterate, find and so on, but is specialized for working with transactions A610.

Specifically, the insert and remove functions may call the notification function on the dependencies A605 that refer to the chain A620.

[00114] A simple A625 list may contain either native objects or objects that inherit from referent A615. An append A630 may implement a list of elements that is determined by traversing a sequence of chain A620 objects. An inverse A635 may contain all those objects that refer to some given object via part instances A525 that reference given part definition A515 (see Figure A5). A filter A640 is a list that refers to a base chain A620, and a condition that determines which of the elements of the base chain A620 are to be considered elements of the filter A620 chain. The mapping A305 has a base chain and has a function that maps each of the elements of the base into some other symbolic A520, and those mapped objects are elements of the mapping A305.

[00115] The design of the system expects that when objects are modified or deleted, or added to or removed from a base chain A620, then the chains A620 that reference those chains A620 are updated accordingly. For a mapping A305 and filter A640, an embodiment may construct a transaction for each member of the list, so that the entire list does not have to be reevaluated each time a change occurs.

[00116] An alternative form of change notification may be used. For example, consider the problem of computing the position of edges of objects a diagram. One solution is to construct constraint objects A655 that express the relationships that need to be maintained, such as the distance between the left or right edges of one object from the corresponding edge of another object. Such patterns can be repeated, and so an embodiment may create layout set A650 objects that create such constraints A655 in standard patterns. Those layout sets A650 can be referenced in an arrangement A260 (see Figure A2).

[00117] An embodiment of the constraint objects A655 may use a specialization of the standard observer design pattern. For example, the position of the edges of graphical objects can each be held on anchor A660 objects. An observer A665 object has a list of anchors A660 that it watches and a constraint A655 that it will notify when the anchor is modified. Unlike the dependency A605, a constraint A655 is not deleted when notification occurs. Observers A665 may be created when a constraint A655 is created, when the constraint A655 is deleted, and when all of its anchors A660 are deleted. When a constraint A655 is notified of an update, it accesses the values of the source anchors A660 it is connected to and computes a new target value, and stores the resulting value on its target anchor A660, which then notifies its

observers A665. The logic that a particular constraint A655 uses to compute the target value is determined by the layout set A650 that is selected by the user. The embodiment may provide a collection of such layout sets A650 for users to choose which may include behaviors such as fixed offsets from a parent or child, offsets computed from the vertical or horizontal size of a child or the maximum size appearing in the list of children. An embodiment may provide more specialized layout sets as needed, such as for example to perform automatic layout of nodes and links in a diagram.

[00118] BUILDING A DIAGRAM

[00119] Figure A7 illustrates the workbench A540, which is a development environment that may be used to create applications. The workbench A540 may be used to create, view, modify, and in some cases, execute symbolic A520 (see Figure A5) objects, which include component definitions A120 (see Figure A1).

[00120] A workbench A540 may contains various types of objects. The term "node" is used to refer to an object that can be moved around by the user. One type of node may be a jig A720. An embodiment may allow a user to add and delete nodes by providing user interface mechanisms such as, for example, menu items. The embodiment may also provide a user interface mechanisms for additional standard operations such as delete, move, copy, paste and so on. Each jig A720 may reference a symbolic A520. Some of the operations may affect the jigs A720 while others affect the objects that the jigs A720 refer to. An embodiment may also build a list of jigs A720 based on some other list, for example the list of objects that are in a particular external file A545, or a directory containing such files. The user interface mechanism may be implemented using the workbench arrangement A705.

[00121] An embodiment may provide user interface elements to create a new instance of a symbolic A520 and assign to that object a reference on a new or existing jig A720. The choices of which classes to present as choices to create instances of may be determined by the embodiment based on the basis object, if any, associated with the workbench A540. If the workbench A540 has a basis, then only those objects that need to be presented would be determined by the part definitions A515 for the model definition A505 associated with the basis need to be presented.

[00122] An embodiment may also provide user interface elements, such as drag and drop, or a hierarchical menu system, for example, to allow the user to select from among existing symbolic A520 to create a reference on a jig A720 to that object. (Several of the following

items appear on Figure A5). The embodiment may present a list of symbolic A520 objects that may include objects in a library provided with the embodiment and also objects loaded from other external files A545 via import A530. An embodiment may restrict the choices that are presented based on the type of the basis object in the same manner as described for creating new objects.

[00123] An embodiment may implement an application by providing a main routine that loads an external file A545 that uses the format described above to import A530. When an embodiment exports A535 a file it may arrange that one of the entries in the exported file A545 results in the creation of a top level controller A700 that references a workbench A540 and a workbench arrangement A740 that will control the presentation of the workbench A540. When the embodiment imports A530 an external file is A545, an embodiment may initiate the selective expansion A115 using that top-level controller A700 as described as for controller A400 (see Figure A4).

[00124] Reference is now made to items on Figure A4 to describe a possible embodiment of items from Figure A7 that have already been described. The top-level controller A700 is an instance of a controller A400, and so it may refer to a view tree A405 with a manager A406 that instructs the system to construct an application window in the host graphics system. That view tree A405 may also hold view tree contents A407 for other user interface elements, such as a top level menu. The following paragraphs describe a possible embodiment of the lower level details of a workbench A540.

[00125] An embodiment of a workbench arrangement A705 may refer to a view tree A405 that contains a view mapping A445 for workbench nodes, such as jigs A720, and also a view mapping A445 for the links that an embodiment may display between the jigs A720, or between lower level connection points within the jigs A720. A view mapping A445 may iterate over the list of nodes in the workbench A540 and may expand an arrangement A260 for each node. The arrangement A260 for each node may refer to a view tree A405 that has view tree contents A407 that determines what is to be shown for each jig A720. Those may include a popup menu, and icons for moving the jig A720, icons for controlling the amount of detail within each jig A720, and a border. An arrangement A260 for the border may refer to a view tree A405 that contains a view single A415 for the border itself and a view reference A440 that switches to a jig arrangement A725 that is constructed as described below, and which is

based on the model definition A505 (see Fig 5.) that is referenced by the basis object of the jig A720.

[00126] An embodiment may use the nesting of these view items A265 and arrangements A260 together with constraint A655 (see Figure A6) and related objects to maintain the size of a border to match the size of its contents. The sequence of arrangements A260 may be used by an embodiment to control which parts of the display appear on top of the other parts, provided that the embodiment always redraws later objects when an earlier object is drawn.

[00127] Reference is now made to further items on Figure A7, using items on Figure A4 that have been previously described. An embodiment may construct a jig arrangement A725 that refers to a model view formation A710. The model view formation A710 may be a view formation A465 that is applied to a model descriptor list A715, which is a list that contains a descriptor A460 for some or all of the part definitions A515 in the model definition A505. An embodiment may construct a model descriptor list A715 as follows. A part definition A515 may have a name that is a descriptor name. An embodiment may get the descriptor name from an annotation in the embodiment source code, if the programming language provides annotations, or it can be an attribute added to a part definition A515 that was imported.

[00128] The descriptor A460 name may be used to find a descriptor A460 with that name by an embodiment that keeps a searchable collection of descriptor objects. An embodiment may use a descriptor A460 that is shared between different part definitions A515 by making a copy of the shared descriptor A460 and then storing the specifics for each part definition A515 on the descriptor A460 copy, possibly including a reference to the part definition A515 itself.

[00129] The jig arrangement A725 that is constructed by an embodiment for a model definition A505 may be cached and reused. That cache may be stored on an object that is associated with a single workbench A540 or shared between multiple workbenches A540 so that the embodiment can provide multiple representations of the same objects in different workbenches A540 and may switch between those representations at run time.

[00130] As described earlier, an embodiment of a view formation A465 may map a list of descriptors A460 into a list of arrangements A260. That pattern maybe applied to this case, so that the embodiment of a model view formation A710 may create mount arrangements A725.

[00131] Reference is now made to further items on Figure A7, using items on Figures 4 and 5 that have been previously described. An embodiment may also create objects called mounts A730 for each descriptor. A mount A730 may be used to store settings that the user has made

so that those settings are exported A535 to an external file A545 so they can be used again when the external file A545 is imported A530. An embodiment may find a mount A730 that corresponds to a part definition A515 by storing the identifier of the part definition A515 on the mount A730. The mounts A730 are associated with jigs A720 to control how information is displayed in the jig A720. An embodiment may build up a jig arrangement A725 using a view tree A405 that contains mount arrangements A735.

[00132] There are several kinds of mount arrangements A735 that an embodiment may provide, depending on the type of data that the part instances A525 associated the corresponding part definition A515 are intended to hold.

[00133] A described earlier, an embodiment may provide a function to construct an arrangement A260 from information stored on a descriptor A460. An embodiment may build mount arrangements A735 by calling such a function to construct an arrangement A260 from the descriptors A460 in the model descriptor list A725. An embodiment may complete that mount arrangement A735 by adding additional information from the descriptor A460, such as a label and/or tooltip text for example, and the view item A265 that will be accessed to provide the details of that part of the display.

[00134] The following paragraphs then describe how an embodiment may construct each mount arrangement A735 from the descriptor A460 obtained from the part definitions A515.

[00135] For part definitions A515 for a primitive type, such as string or integer for example, the descriptor A460 can provide a reference to a view single A415 that causes a text edit graphic object to be created. The response A235 for such a type would store the input from the text edit field on a part instance A525 that is on the basis of the mount A730. The part instance A525 to store the value on may be found by matching its reference to its part definition A515 to the part definition that may be stored on the descriptor A460.

[00136] For part definitions A515 that describe a value that is a referent A615, an embodiment may construct a mount arrangement A735 that will capture an expression that evaluates to the referenced object. An embodiment may provide several formats, including a textual reference, or a link object that is connected to a graphic object that may be generated from another jig A720. Another case is when the referenced object is local to, and owned by, a specific part instance A525. In that case, the embodiment may create a nested jig A740 that is associated with the mount A730 that corresponds to that part definition A515. The embodiment may then create a jig arrangement A725 for the nested jig A740 in a similar

manner as for a jig A720 located immediately under a workbench A540. In that case, the jig arrangement A725 may be located within the mount arrangement A735 that displays the parent part instance A515.

[00137] For part definitions that describe a list type value, an embodiment may construct a mount arrangement A735 that holds a reference to a workbench arrangement A705 that is specialized for a nested workbench A745. An embodiment of a nested workbench A745 may have a mapping A305 from the members of the list value to jigs A720 that reference those list members.

[00138] The graphic objects that are used to display the jigs A720 may be positioned using constraints A655 so that they form a compact list, or an embodiment may obtain the position to display each jig A720 from a position type part instance A525 associated with that jig A720. When the user drags the mouse over a graphic object associated with the jig A720, the embodiment may store the updated position of the displayed object back on the position type part instance A525 of the jig A720. An embodiment may provide code to assign a default position to the jigs A720 by scanning the list of jigs A720 to find a vacant space to place a new jig A720.

[00139] Reference is now made to previously described items on Figure A6, using items on Figures 4 and 7 that have also been previously described. If an embodiment provides links, for example as references from a part instance A525 to another symbolic A520, it may do so by using a view mapping A445 in the view tree A405 referenced by the workbench arrangement A705, as described above. An embodiment may build a chain A620 of the links that need to be displayed by building an inverse A635 that contains all the references from the objects displayed by each of the mounts A735 where a link has been stored. An embodiment may use a mapping A305 to generate an arrangement A260 for each of these links. The embodiment of each link arrangement A260 may reference a view single A415 that references a behavior instance A420 that will create a graphic object that displays a link. The link arrangement A260 may include a user interface element, such as menu item, that allows the link to be deleted. The layout set A650 for the link may construct constraints A655 that assign the position of each end of the link to the location assigned to the mount arrangement A735 that is associated with the part definition A515 being displayed. An embodiment may find the display object, and its location, for that arrangement A260 by using a register table.

[00140] An embodiment may provide a user interface element, such as a menu item, on the workbench arrangement A705 that invokes export A535 so that a user can save work for reloading later or for using the saved external file A545 in an application. That exported file A545 may contain the workbench A540, the jigs A720, mounts A730, nested jigs A740, and nested workbenches A745 described above. The workbench A540 and associated objects may be stored in a different external file A545 from that used for other internal objects A100.

[00141] When the symbolic A520 objects displayed in a workbench correspond to component definitions A120 and related objects, the embodiment may provide an option to omit the workbench A540 objects and save just the symbolic A520 objects referenced by them to form an external file A545 that is intended to be an application file. An embodiment may also have a user interface element that invokes a code generator that produces source or machine executable code corresponding to those component definitions A120 and related objects. An embodiment may produce the source code by following the same sequence described here for interpretive execution but substituting emitting source code function and variable declarations and executable statements to an output stream instead of executing the steps. An embodiment may produce executable code by for example, producing source code as described and then invoking a compiler on the resulting file.

[00142] An embodiment may choose to use a commercially available graphics toolkit to provide the display objects or use a custom developed one. The types of display objects that may be needed depend on the details of the user interface desired for the embodiment. Typically the graphic elements needed for such an application include a container that provides scrolling and zooming, a border, text editing, icons, links, and menus. The interface to those classes may be represented in an embodiment as described on Figure A5, and the values on the instances of those objects may be controlled as described in Figure A3.

[00143] Figure A8 illustrates an exemplary application creation routine in accordance with one embodiment. An embodiment of an application built using this example may appear as shown in Figure A9, which contains a hierarchical diagram. The exact nature of the steps described below depends on the form of the embodiment. One possibility is to use a text editor to build an external text file A545 (see Figure A5). Another possibility is to use a workbench A540 (see Figure A5), which may be a software application where the user constructs objects via a graphical user interface. The implementation of such a system was described in Figure A7. In either approach, the order of steps may be flexible, and building an application may be

an iterative process where the user builds the application incrementally and views the results so far.

[00144] In block A805 a top-level controller A400 is defined. As described in Figure A4, an embodiment may use a controller A400 to hold the arrangement A260 that is used to construct the component instances A105 (see Figure A1) that form the application. The top-level controller A400 may refer to the first arrangement A260 described below.

[00145] In block A810, data definitions are created. (See Figure A5, discussed above.) In an exemplary embodiment, there may be a model definition A505 for the basis object for the diagram as a whole. It may be given a name and two part definitions A515. One part definition A515 is for the list of references to nodes and the other is for the list of links on the diagram. There may be an additional model definition A505 for the link objects. The model definition A505 for the nodes may be extension of the model definition A505 for the diagram as a whole, so that nodes can contain nodes hierarchically, and may additionally have a part definition A515 for the position.

[00146] In block A815, one or more diagram arrangements may be created. (See Figure A4, discussed above.) In an exemplary embodiment, the first arrangement A260 may refer to a top level view tree A405. A view tree A405 may take a manager A406, which in this case may refer to a view single A415 that refers to a component type A200 that describes a main window object in a host graphics system. The top level view tree A405 would have view tree contents A407 that in this example may be the node and link arrangements, which are created in block A820 (see Figure A8).

[00147] The arrangements A260 for the diagram nodes and links may each refer to a view mapping A445. Each view mapping A445 may take an expression that refers the corresponding list type part definition A515, described in block A810 above. Each view mapping A445 may also take an arrangement A260 that may be mapped for each element in the list. In this example, the arrangement A260 for links may refer to a view single A415 that refers to a component type A200 for a graphic object that appears as an arrow. For nodes in a non-hierarchical diagram, the arrangement would be very similar, but the component type A200 would refer to a box graphic.

[00148] Since Figure A9 shows a hierarchical diagram A900, an extra layer is needed so that nodes may contain nodes in a way that reflects the structure of the data model. In this case, the arrangement A260 for the nodes may refer to a view tree A405 whose view tree contents

A407 may include the box view single A415 mentioned above and also an arrangement A260, which refers again to the previous view mapping A445 for the nodes. This cycle of references in the node arrangements A260 makes the diagram hierarchical.

[00149] As described before, an embodiment of a view mapping A445 may set the basis to be each successive object in the list that is being mapped. The position of the box may be given as an expression that accesses the basis from the controller A400 and gets a position from that basis using the position part definition A515 created earlier. The arrangement A260 for the nodes may refer to a layout set A650 (see Figure A6) that provides a fixed size of the nodes and also adds to its position value the position of the parent object. That will cause a nested node to appear relative to its parent object. The layout set A650 for the link arrangement A260 may reference a layout set A650 that obtains the positions of the end points of the link from the positions obtained from the register table A310 (see Figure A3). The entries in the register table A310 for the nodes may be created using a component argument A230 (see Figure A2) that is added to the node arrangement A260.

[00150] In block A825, input data may be prepared. (See Figure A5, discussed above.) For the examplary embodiment shown in Figure A9, there would be one symbolic A520 each for the diagram, the five nodes, and the three links. Each symbolic A520 would reference the corresponding model definition A505 constructed earlier. The symbolic A520 for the diagram basis would have part instances A525 for both the list of nodes and the list of links. The value of those lists may contain expressions that refer to the node and link symbolic A520 objects being defined here. The node symbolic A520 objects would each have a part instance A525 that holds the position value. The symbolic A520 for each of the nodes may also contain a part instance A525 for its list of reference to nested nodes.

[00151] In block A830, the application A830 may be run. For a graphical workbench A540, as illustrated in Figure A5 and discussed above, the user may simply push a button in the user interface, and that button may be defined in the embodiment to expand the top-level controller A400 (see Figure A4). For a text-based embodiment, the user may run a base program provided by the embodiment that imports A530 the external file A545 containing the example. When that program encounters the define top-level controller A805 object, it may run the application by the expanding the top- level controller.

[00152] Figure A9 shows what an embodiment may look like when running the example created by the routine illustrated in Figure A8. The nodes A910A-E shown in hierarchical

VS

diagram A900 correspond to the node objects and the links A905A-C correspond to the link objects in the data model, for example. A larger application may include additional arrangements for text and graphic annotations, a menu system, multiple diagrams, operations to manipulate the objects, an interface to external data source and events, etc.

[00153] Although specific embodiments have been illustrated and described herein, a whole variety of alternate and/or equivalent implementations may be substituted for the specific embodiments shown and described without departing from the scope of the present disclosure. This application is intended to cover any adaptations or variations of the embodiments discussed herein.

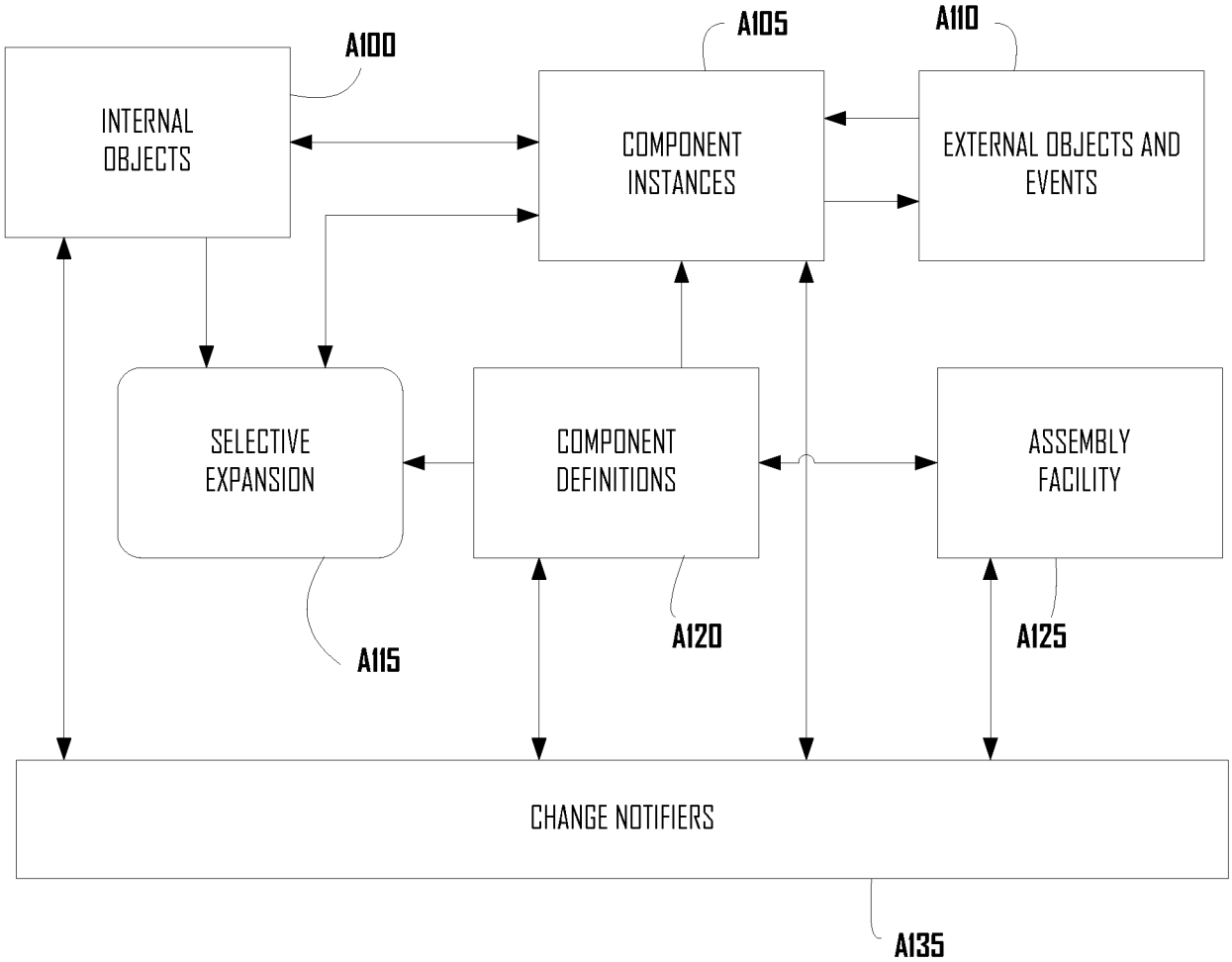
A100
INTERNAL OBJECTS
A105
COMPONENT INSTANCES
A110
EXTERNAL OBJECTS AND EVENTS
SELECTIVE EXPANSION
COMPONENT DEFINITIONS
ASSEMBLY FACILITY
A115
A120
A125
CHANGE NOTIFIERS
A135

Fig. A1

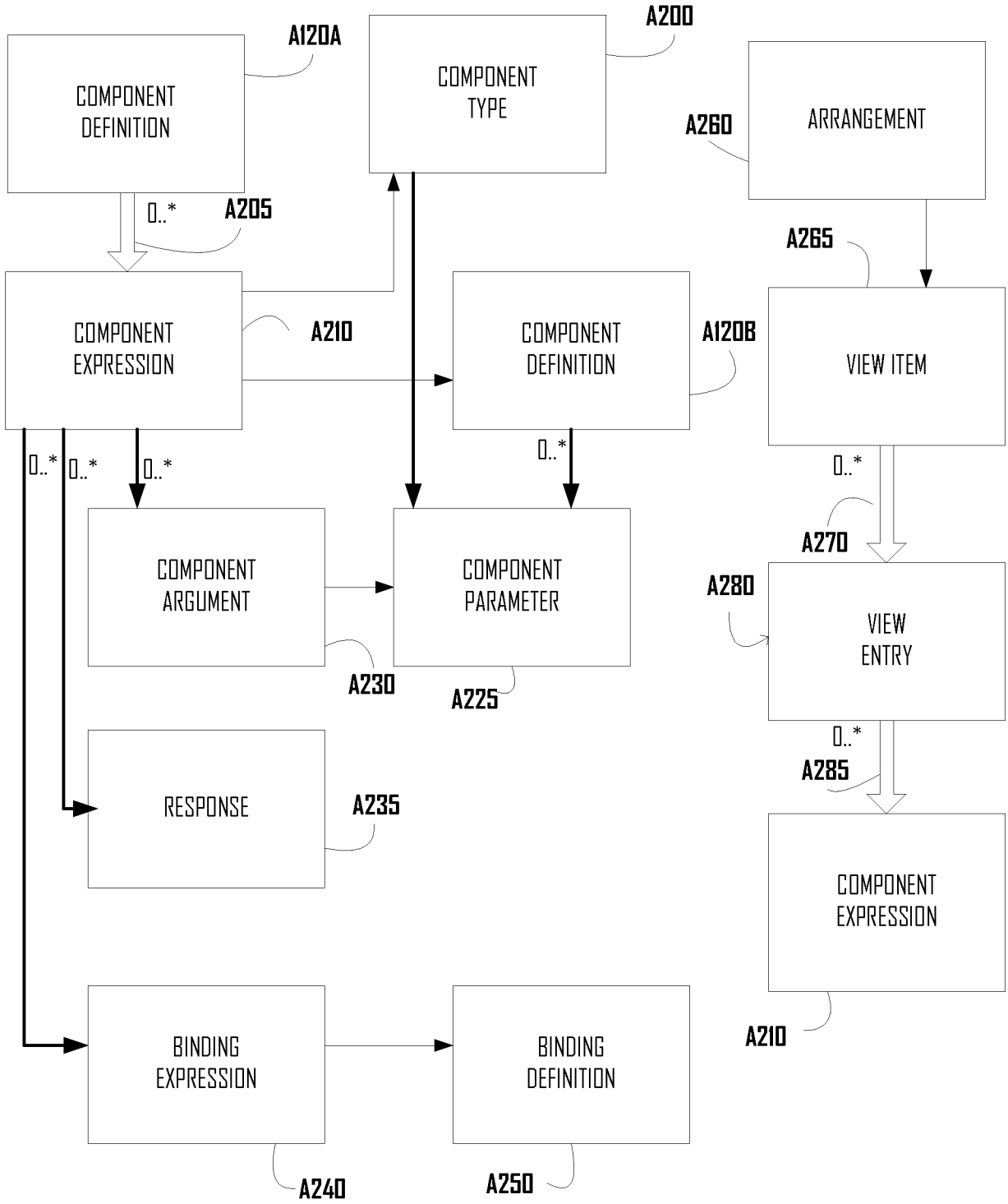
A120A
COMPONENT DEFINITION
A200
COMPONENT TYPE
A260
ARRANGEMENT
0..*
A205
A265
COMPONENT EXPRESSION
A210
COMPONENT DEFINITION
A120B
VIEW ITEM
0..*
0..*
0..*
0..*
A270
COMPONENT ARGUMENT
COMPONENT PARAMETER
A280
VIEW ENTRY
A230
A225
0..*
A285
RESPONSE
A235
COMPONENT EXPRESSION
BINDING EXPRESSION
BINDING DEFINITION
A210
A240
A250

Fig. A2

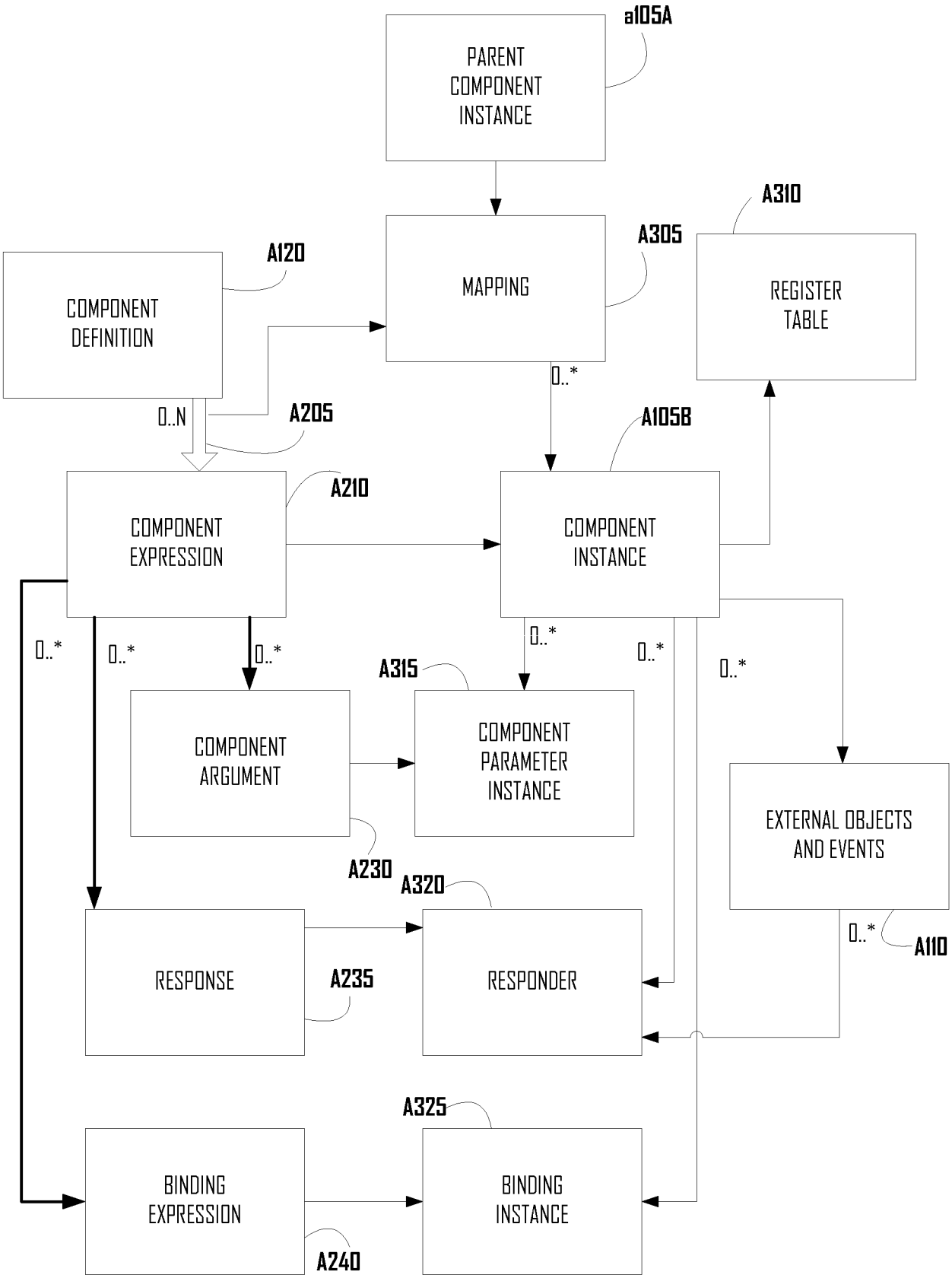
a105A
PARENT COMPONENT INSTANCE
A310
A305
A120
COMPONENT DEFINITION
MAPPING
REGISTER TABLE
0..*
0..N
A205
A105B
A210
COMPONENT EXPRESSION
COMPONENT INSTANCE
0..*
0..*
0..*
0..*
0..*
0..*
A315
COMPONENT ARGUMENT
COMPONENT PARAMETER INSTANCE
EXTERNAL OBJECTS AND EVENTS
A230
A320
0..*
A110
RESPONSE
A235
RESPONDER
A325
BINDING EXPRESSION
BINDING INSTANCE
A240

Fig. A3

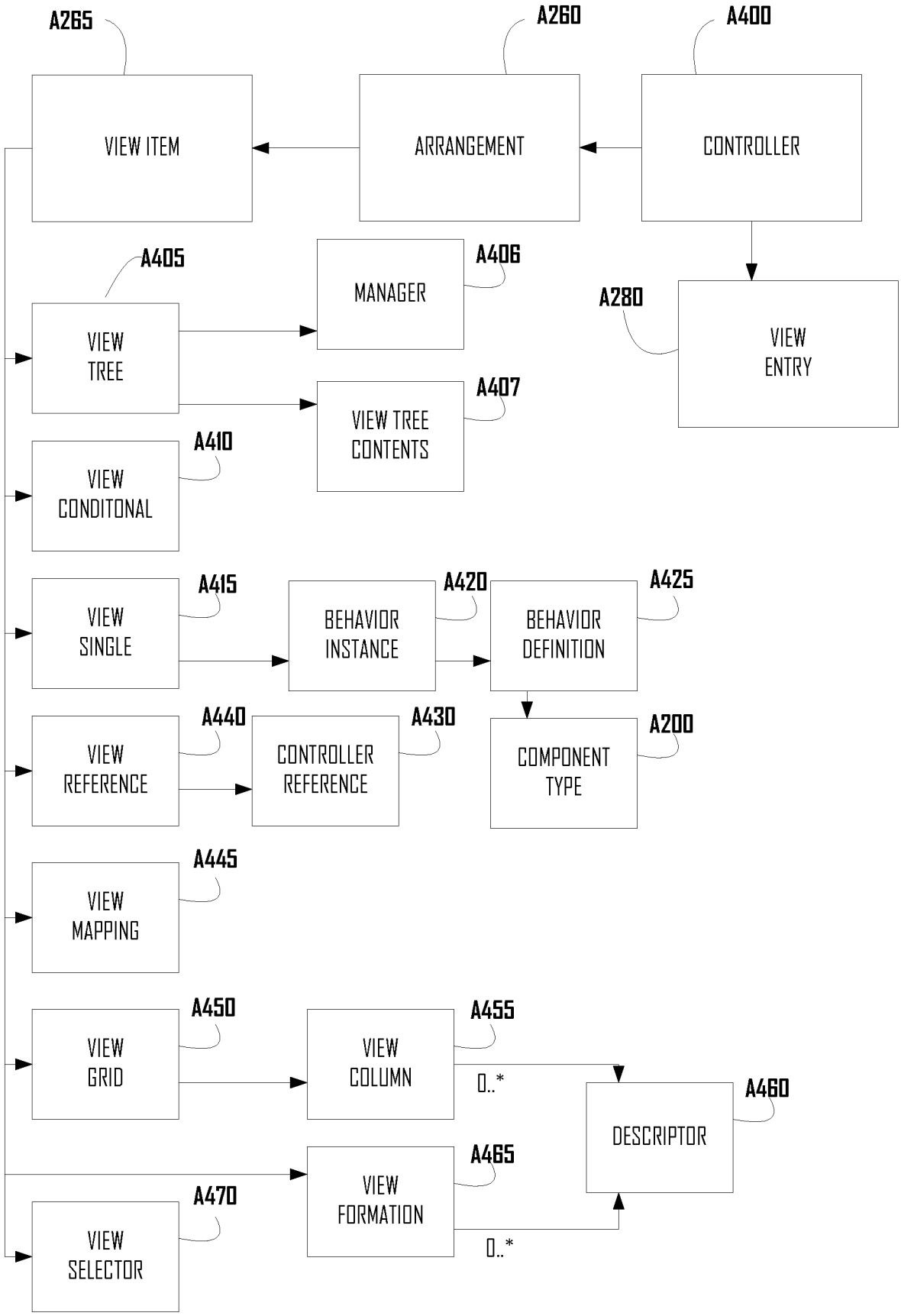
A265
A260
A400
VIEW ITEM
ARRANGEMENT
CONTROLLER
A405
A406
MANAGER
A280
VIEW TREE
VIEW ENTRY
A407
VIEW TREE CONTENTS
A410
VIEW CONDITONAL
A415
A420
A425
VIEW SINGLE
BEHAVIOR INSTANCE
BEHAVIOR DEFINITION
A440
A430
A200
VIEW REFERENCE
CONTROLLER REFERENCE
COMPONENT TYPE
A445
VIEW MAPPING
A450
A455
VIEW GRID
VIEW COLUMN
0..*
A460
DESCRIPTOR
A465
VIEW FORMATION
A470
0..*
VIEW SELECTOR

Fig. A4

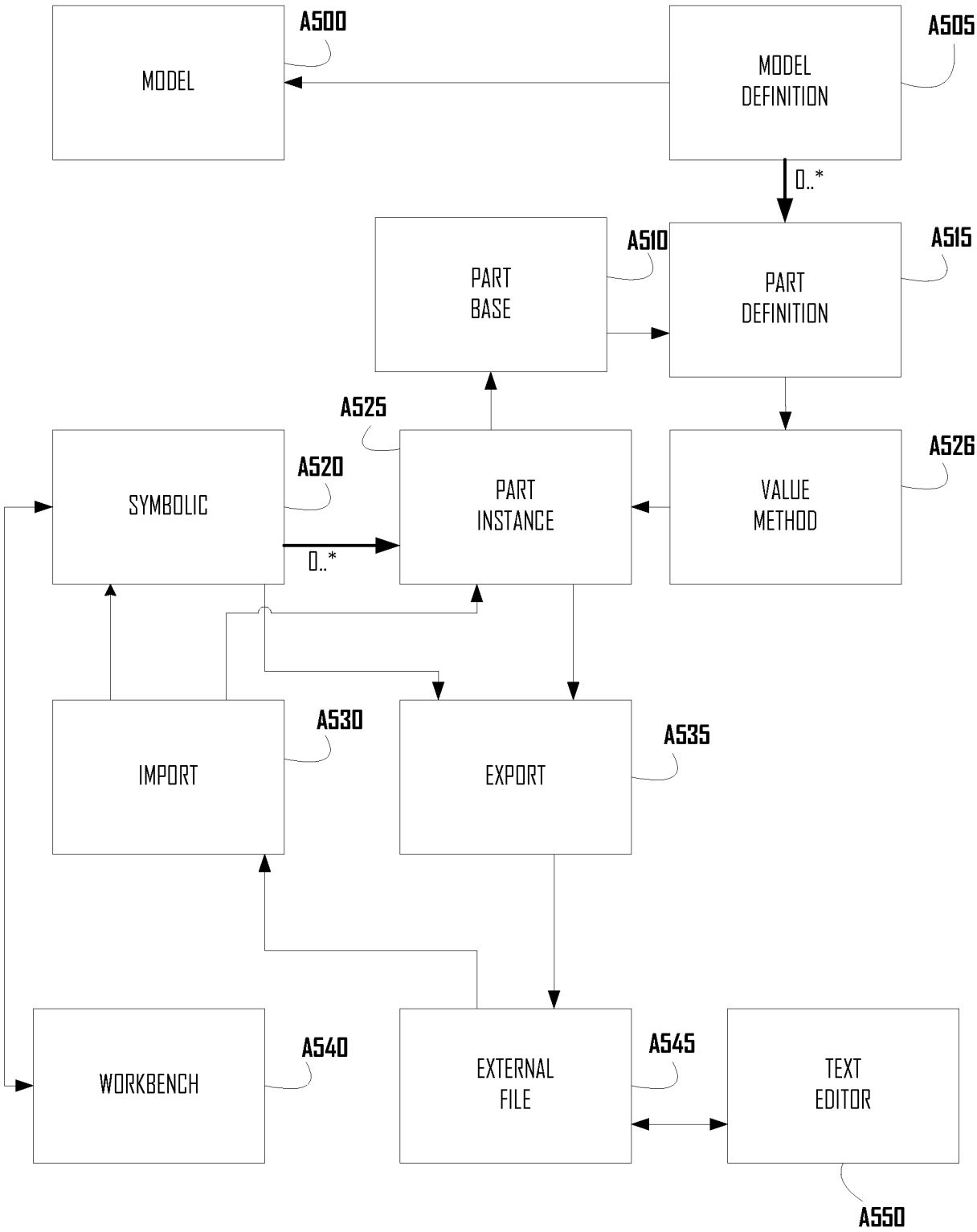
MODEL
A500
MODEL DEFINITION
A505
0..*
PART BASE
A510
PART DEFINITION
A515
A525
SYMBOLIC
A520
PART INSTANCE
VALUE METHOD
A526
0..*
IMPORT
A530
EXPORT
A535
WORKBENCH
A540
EXTERNAL FILE
A545
TEXT EDITOR
A550

Fig. A5

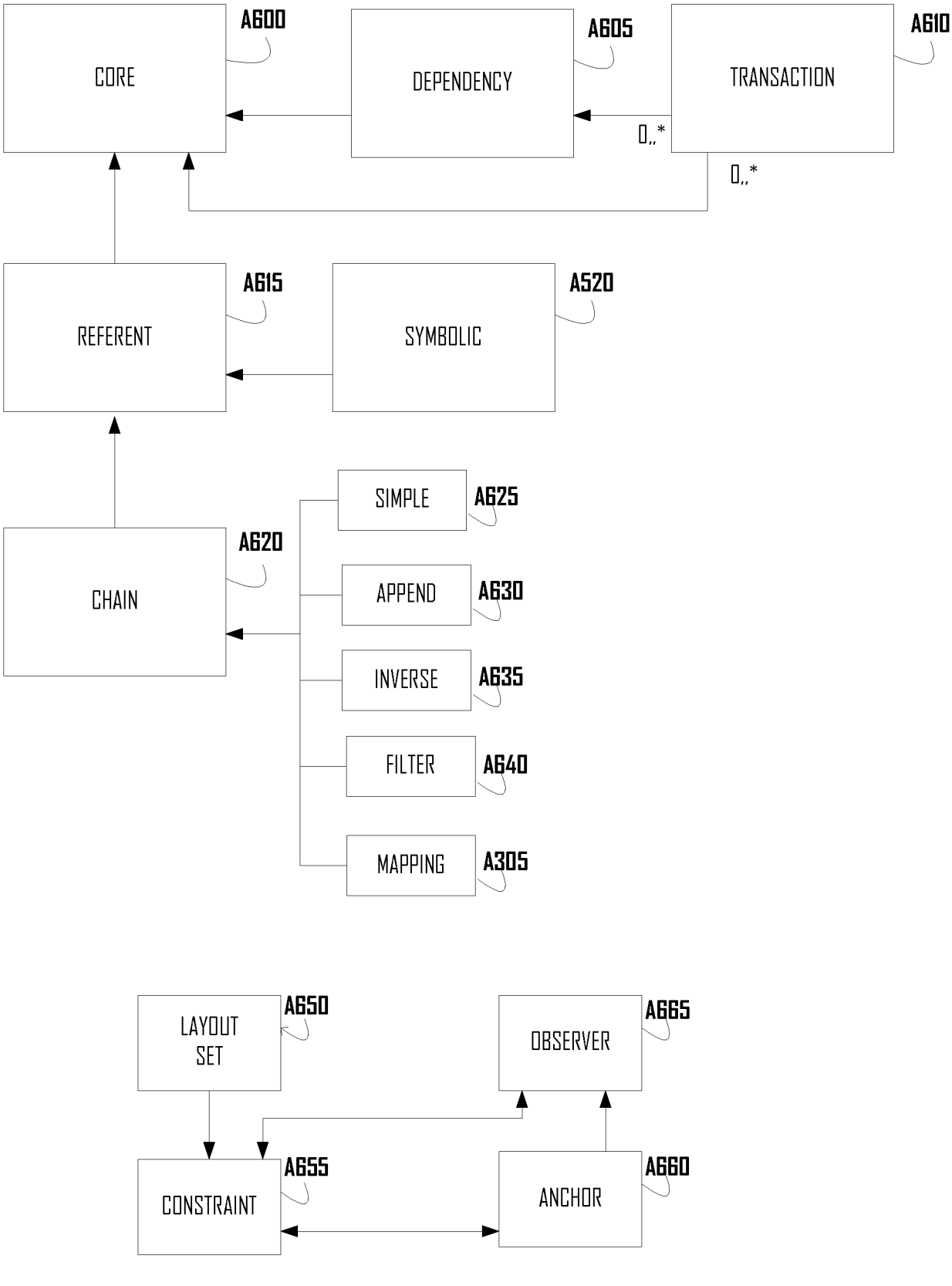
CORE
A600
DEPENDENCY
A605
TRANSACTION
A610
0..*
0..*
REFERENT
A615
SYMBOLIC
A520
CHAIN
A620
SIMPLE
A625
APPEND
A630
INVERSE
A635
FILTER
A640
MAPPING
A305
LAYOUT SET
A650
OBSERVER
A665
CONSTRAINT
A655
ANCHOR
A660

Fig. A6

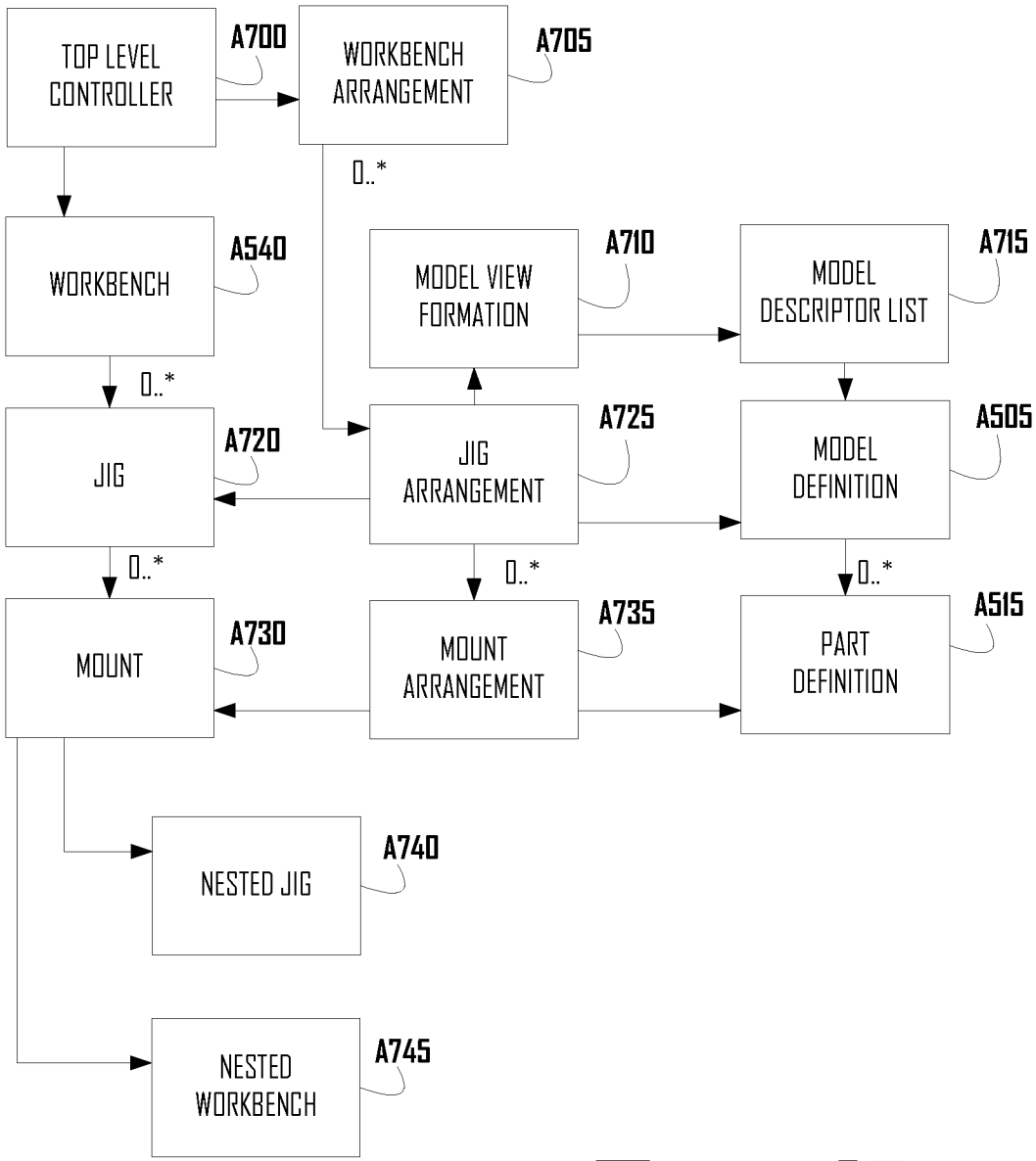
TOP LEVEL CONTROLLER
A700
WORKBENCH ARRANGEMENT
A705
0..*
WORKBENCH
A540
MODEL VIEW FORMATION
A710
MODEL DESCRIPTOR LIST
A715
0..*
JIG
A720
JIG ARRANGEMENT
A725
MODEL DEFINITION
A505
0..*
0..*
0..*
MOUNT
A730
MOUNT ARRANGEMENT
A735
PART DEFINITION
A515
NESTED JIG
A740
NESTED WORKBENCH
A745

Fig. A7

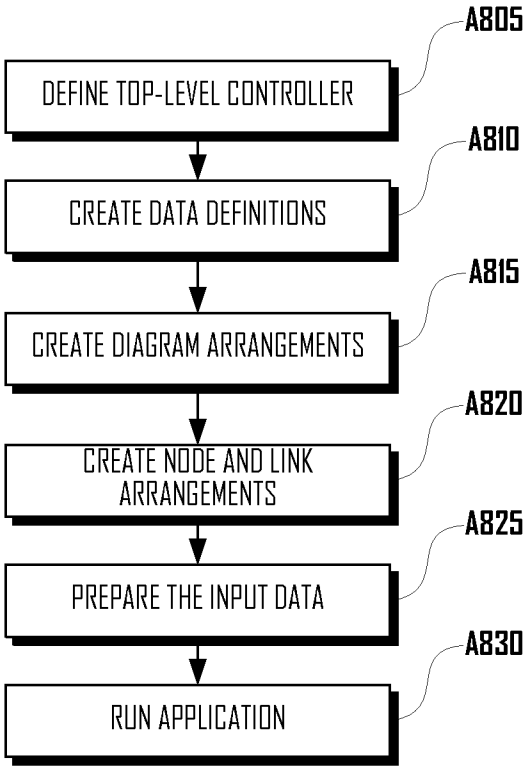
A805
DEFINE TOP-LEVEL CONTROLLER
A810
CREATE DATA DEFINITIONS
A815
CREATE DIAGRAM ARRANGEMENTS
A820
CREATE NODE AND LINK ARRANGEMENTS
A825
PREPARE THE INPUT DATA
A830
RUN APPLICATION

Fig. A8

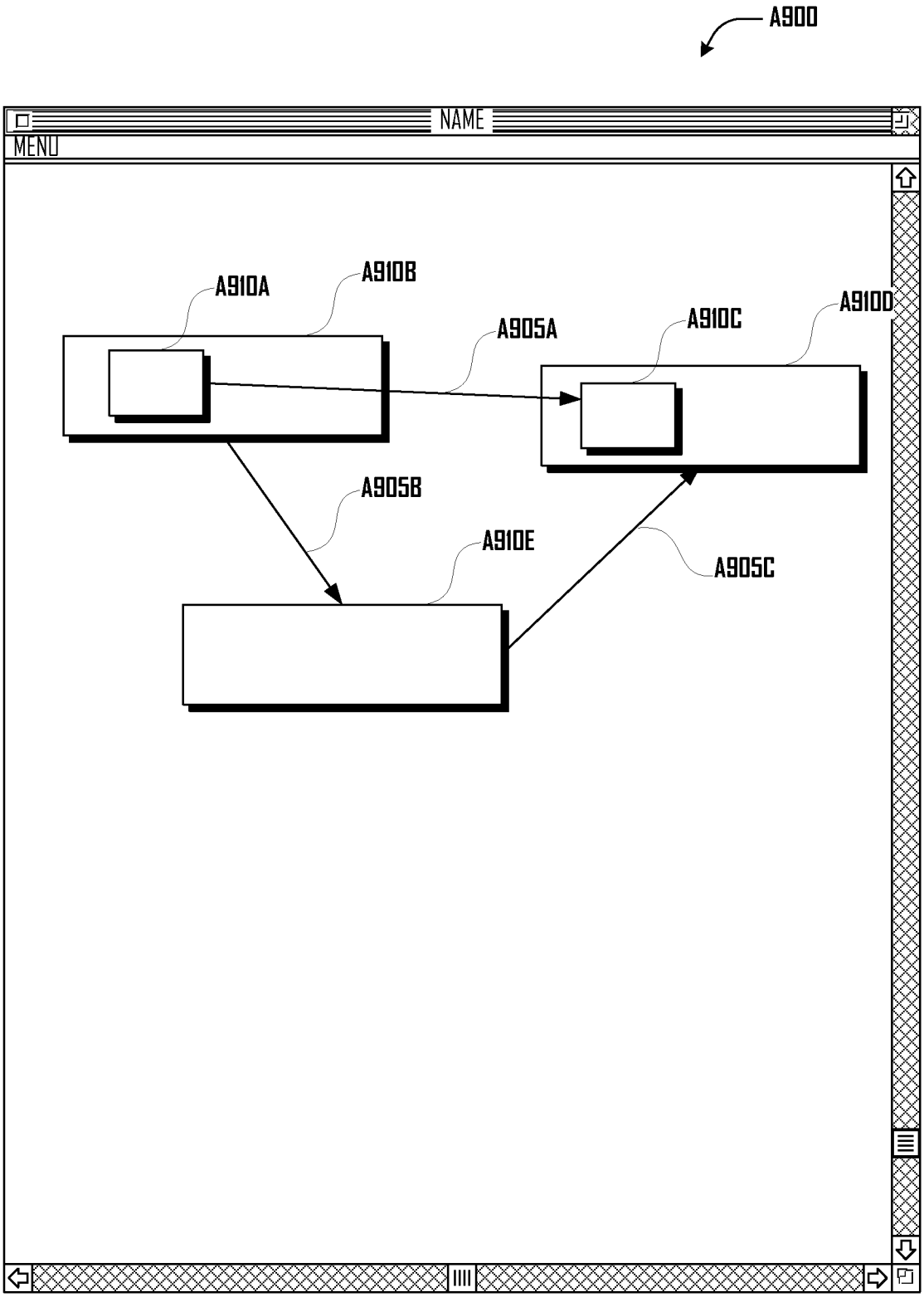
A900
NAME
MENU
A910A
A910B
A905A
A910C
A910D
A905B
A910E
A905C

Fig. A9

APPENDIX B

AUTOMATIC GRAPH LAYOUT

[Para 01] In various embodiments, the automatic arrangement of nodes on a visualization graph may be controlled according to various methods. For example, in some embodiments, nodes can be positioned manually by the user or automatically, in a tree structure or as a directed graph.

[Para 02] As the term is used in Appendix B, a node that is the subject of an explosion operation (e.g., a "parent" node, as referred to in the body of this application) is referred to as an "exploded" node. In other words, a node whose results have been exploded into one or more "child" or "explosion" nodes. Because Appendix B deals with general graph placement (not limited to tree-structures), the terms "parent" and "child" may be misleading, and so they are not used.

[Para 03] When node is exploded, and some or all of its explosion nodes are themselves exploded, the result is called a tree. The need for more general graph placement arises when multiple explosions can refer to the same node. For example, the results for two exploded nodes may contain some rows that have the same values. In some embodiments, the user may be able to select whether the system is to show two different nodes or to have a single shared node that has links from the two different exploded nodes. A graph that contains shared nodes is no longer a tree and so it needs a more generalized method to determine the node positions.

[Para 04] A variety of tree or non-tree diagrams can be quickly created by choosing several settings on a query or node. In one embodiment, these settings include some or all of the following settings:

- the default query to be selected for each explosion node that results from that query;
- the default linkage to be selected for the link to each explosion node that results from that query;
- an indication that each explosion node is also to be exploded, either recursively or just one level;
- an indication that nodes are to be shared if an existing node refers to the same set of query result values;
- an indication of whether the initial exploded node is to be linked to its explosion nodes;

- the algorithm that is selected to be used to perform the layout of the nodes and links
- setting values used by the selected layout algorithm.

[Para 05] Some of these setting values are commonly used together and so the user interface can present a list of collections of settings, for example for a tree or specific styles of diagrams.

[Para 06] A general graph is constructed when the user has indicated that the each explosion node is itself to be exploded and the result nodes are shared. This results in a potentially large collection of nodes with an arbitrary pattern of directed links. The links in such the diagram have the same behavior of query parameters and linkages as described previously.

Methods for Node Placement

[Para 07] In most data visualization tools, when results are displayed graphically, the position of the nodes is often obtained from numeric values contained in the results. In such a case, as a part of the query, the user is presented with selections to indicate which columns are to be used for various aspects of the visualization. These include a large number of values such as x-axis, y-axis, z-order, node width, node height, color, edge style, and label text.

[Para 08] Alternatively, the positioning can be assigned relatively algorithmically, with the goal being to present the nodes in an arrangement that helps the user to see the overall structure of the link relationships between the nodes. In the case where the user selects an option to cause explosion nodes to not be shared, then the graph is a tree and a simple set of parameters can be used to shape the tree, such as direction of explosion, spread angle, the orientation and curvature of the line where the explosion nodes are placed, and the like. In one embodiment, a layout algorithm for such a tree diagram can walk the tree and assign positions to the nodes so that they do not overlap. Such a layout algorithm can do this at level of the tree by computing the distance needed at the specified angles to prevent overlap of the sub-trees of each node in the tree. Alternatively, the user can specify a distance and have the system compute the angles, or the user can specify both manually.

[Para 09] When the graph does not have a tree structure, a more complex algorithm is needed. A commonly-used approach is called a force-directed algorithm. The basic idea of force-directed layout is to treat each node as a particle that experiences forces, and the algorithm moves the nodes under those forces until an equilibrium is reached. An arbitrary initial position is assigned to each node. The links generally have an attractive force on the nodes at either end that is a function of the length of the link. It is also very helpful if all the forces are assigned in a balanced manner to avoid problems with drift and spinning; i.e., each time a force is applied to one node, an opposite force is assigned to another. A generally repulsive force between nodes is determined also as a function of the distance between the nodes.

[Para 10] There may also be a force to a center point to keep the nodes from drifting off. The position of the center point can be assigned in the same way as described above for a tree type graph, in order to allow a mixed diagram with both tree and graph structured parts. A sub-tree or other independently arranged group of nodes can be treated as a node for the purposes of this algorithm.

[Para 11] The nodes may be arranged into clusters, such that there are no links between different clusters. The system sums the areas of the nodes in each cluster and performs a force-directed algorithm on the clusters as described above.

[Para 12] In some embodiments, node overlap may be avoided by scaling the resulting diagram up, holding node size constant, until their positions no longer overlap. In other embodiments, node overlap may be avoided by extending the standard force-directed algorithm described above as follows.

[Para 13] The force between nodes may be adjusted to reflect the size needed for each node. A scale factor may be applied to the forces to have the node positions reach a desired scale. The scale factor may be determined by finding an approximate equilibrium, and then adjusting the scale in the direction needed for the current position to become closer to the goal scale. Separate goal scales may be maintained for between clusters and within clusters.

[Para 14] The system starts out with a goal of a diagram with a lot of space between nodes and clusters. It initially allows nodes to overlap in order to find a general arrangement. Once a non-overlapping arrangement is achieved at a large scale, the

system then enters into a phase where it does not allow node overlap. In this phase, the motion of each node is constrained so that their motion stops at the point where contact is made. An additional parameter is added to the size of the node so that the point where the stop occurs actually leaves a margin around each node. This is for diagram clarity and to allow links to be routed between nodes. The system repeats the above steps with smaller and smaller goal sizes for the diagram. When there is no more movement of the nodes because there is no more space between them, the algorithm stops.

[Para 15] The above steps may result in a diagram where the nodes are relatively packed, but the structure of the data may not be sufficiently clear. To make the structure clearer, the following changes may be made.

[Para 16] Instead of varying the x and y position directly, there is an additional object called a "bar." Each bar holds a single x or y value. Initially each node gets its own two bars for its x and y values. When the diagram is at a large size, the system choose nodes to share bars for either x or y. The sharing can be determined in various ways, such as if the current position of two nodes is already close to aligned, or a series of nodes have links to the same node and so they should share a x or y bar (chosen based on which way they are currently position), or if a series of nodes are connected in a line. Additional shared bars may also be used to achieve a grid or tree arrangement for some or all the nodes in a cluster. The same force-directed logic described above is used, but forces on the nodes are combined onto the shared bars, and the values of the bars are adjusted instead of the nodes directly. The movement of each bar is constrained by a collision of one of the nodes on the bar with another node, so the bars have to be moved one by one. When two bars have collided, the forces generated on one may be transferred to the bar it has collided with. As the process continues, those bars may become separated and connected many times until all the forces have been effectively canceled out when each bar is pressing up against another bar and can move no further.

[Para 17] The simplest way to route the links is in a straight line between the centers of the nodes. The user can also choose an algorithm that will route the links orthogonally around nodes, perhaps with rounded corners. Various algorithms for orthogonal link placement may be employed once the node positions have been determined.

Claims:

Claim 1. A computer-implemented method for visualizing the structure of data in a database via a plurality of reusable parametric queries and a plurality of graphical nodes at a visualization computer having a graphical user interface ("GUI"), each reusable parametric query comprising a partial query specification, an output field, and an input parameter, each graphical node corresponding to an instance of a reusable parametric query and comprising graphical controls corresponding to the partial query specification of the query instance, the input parameter of the query instance, and one or more result rows of the query instance, the method comprising:

defining, by the visualization computer, a parent reusable parametric query and a child reusable parametric query;

obtaining, by the visualization computer, a parent-parameter value for the input parameter of an instance of said parent reusable parametric query ("parent instance"), said parent-parameter value completing the partial query specification of said parent instance;

executing, by the visualization computer, said parent instance according to said parent-parameter value to obtain a plurality of parent-result rows, each comprising a data element corresponding to the output field of said parent instance;

displaying in the GUI a parent graphical node corresponding to said parent instance; and

for each of said plurality of parent-result rows, performing, by the visualization computer, an explosion operation comprising:

linking the output field of said parent reusable parametric query to the input parameter of said child reusable parametric query;

obtaining a child-parameter value according to the linked output field of the current parent-result row, said child-parameter value completing the partial query specification of said current child instance;

automatically executing said current child instance to obtain at least one child-result row; and

displaying in the GUI a current-child graphical node corresponding to said current child instance, and a graphical linkage control indicating linkage between said current parent-result row and said current-child graphical node.

Claim 2. The method of Claim 1, wherein obtaining said parent-parameter value comprises:

performing a specification query to obtain a specification-result row comprising a specification-result field; and

linking said specification-result field to said parent input parameter of said parent reusable parametric query;

Claim 3. The method of Claim 2, wherein said specification query comprises a reusable parametric query, the method further comprising displaying in said GUI a specification graphical node corresponding to said specification query.

Claim 4. The method of Claim 3, wherein said specification graphical node is automatically created via said explosion operation.

Claim 5. The method of Claim 2, wherein said specification query comprises a reusable parametric query whose input parameter comprises literal data obtained via said GUI.

Claim 6. The method of Claim 1, wherein said parent reusable parametric query further comprises a linkage indication indicating a reusable linkage linking the output field of said parent reusable parametric query to the input parameter of said child reusable parametric query.

Claim 7. The method of Claim 1, wherein said explosion operation further comprises automatically laying out said child node with respect to its peer nodes in said GUI.

Claim 8. A computing apparatus comprising a graphical user interface ("GUI"), a processor, and a memory having stored thereon instructions that when executed by the processor, perform a method for visualizing the structure of data in a database via a plurality of reusable parametric queries and a plurality of graphical nodes, each reusable parametric query comprising a partial query specification, an output field, and an input parameter, each graphical node corresponding to an instance of a reusable parametric query and comprising graphical controls corresponding to the partial query specification of the query instance, the input parameter of the query instance, and one or more result rows of the query instance, the method comprising:

defining a parent reusable parametric query and a child reusable parametric query;

obtaining a parent-parameter value for the input parameter of an instance of said parent reusable parametric query ("parent instance"), said parent-parameter value completing the partial query specification of said parent instance;

executing said parent instance according to said parent-parameter value to obtain a plurality of parent-result rows, each comprising a data element corresponding to the output field of said parent instance;

displaying in the GUI a parent graphical node corresponding to said parent instance; and

for each of said plurality of parent-result rows, performing an explosion operation comprising:

linking the output field of said parent reusable parametric query to the input parameter of said child reusable parametric query;

obtaining a child-parameter value according to the linked output field of the current parent-result row, said child-parameter value completing the partial query specification of said current child instance;

automatically executing said current child instance to obtain at least one child-result row; and

displaying in the GUI a current-child graphical node corresponding to said current child instance, and a graphical linkage control indicating linkage between said current parent-result row and said current-child graphical node.

Claim 9. The apparatus of Claim 8, wherein obtaining said parent-parameter value comprises:

performing a specification query to obtain a specification-result row comprising a specification-result field; and

linking said specification-result field to said parent input parameter of said parent reusable parametric query;

Claim 10. The apparatus of Claim 9, wherein said specification query comprises a reusable parametric query, the method further comprising displaying in said GUI a specification graphical node corresponding to said specification query.

Claim 11. The apparatus of Claim 10, wherein said specification graphical node is automatically created via said explosion operation.

Claim 12. The apparatus of Claim 9, wherein said specification query comprises a reusable parametric query whose input parameter comprises literal data obtained via said GUI.

Claim 13. The apparatus of Claim 8, wherein said parent reusable parametric query further comprises a linkage indication indicating a reusable linkage linking the output field of said parent reusable parametric query to the input parameter of said child reusable parametric query.

Claim 14. The apparatus of Claim 8, wherein said explosion operation further comprises automatically laying out said child node with respect to its peer nodes in said GUI.

Claim 15. A non-transient computer-readable storage medium having stored thereon instructions that when executed by a processor, perform a method for visualizing the structure of data in a database via a plurality of reusable parametric queries and a plurality of graphical nodes, each reusable parametric query comprising a partial query specification, an output field, and an input parameter, each graphical node corresponding to an instance of a reusable parametric query and comprising graphical controls corresponding to the partial query specification of the query instance, the input parameter of the query instance, and one or more result rows of the query instance, the method comprising:

defining a parent reusable parametric query and a child reusable parametric query;

obtaining a parent-parameter value for the input parameter of an instance of said parent reusable parametric query ("parent instance"), said parent-parameter value completing the partial query specification of said parent instance;

executing said parent instance according to said parent-parameter value to obtain a plurality of parent-result rows, each comprising a data element corresponding to the output field of said parent instance;

displaying in a graphical user interface ("GUI"), a parent graphical node corresponding to said parent instance; and

for each of said plurality of parent-result rows, performing an explosion operation comprising:

linking the output field of said parent reusable parametric query to the input parameter of said child reusable parametric query;

obtaining a child-parameter value according to the linked output field of the current parent-result row, said child-parameter value completing the partial query specification of said current child instance;

automatically executing said current child instance to obtain at least one child-result row; and

displaying in said GUI a current-child graphical node corresponding to said current child instance, and a graphical linkage control indicating linkage between said current parent-result row and said current-child graphical node.

Claim 16. The storage medium of Claim 15, wherein obtaining said parent-parameter value comprises:

performing a specification query to obtain a specification-result row comprising a specification-result field; and

linking said specification-result field to said parent input parameter of said parent reusable parametric query;

Claim 17. The storage medium of Claim 16, wherein said specification query comprises a reusable parametric query, the method further comprising displaying in said GUI a specification graphical node corresponding to said specification query.

Claim 18. The storage medium of Claim 17, wherein said specification graphical node is automatically created via said explosion operation.

Claim 19. The storage medium of Claim 16, wherein said specification query comprises a reusable parametric query whose input parameter comprises literal data obtained via said GUI.

Claim 20. The storage medium of Claim 15, wherein said parent reusable parametric query further comprises a linkage indication indicating a reusable linkage linking the output field of said parent reusable parametric query to the input parameter of said child reusable parametric query.

Claim 21. The storage medium of Claim 15, wherein said explosion operation further comprises automatically laying out said child node with respect to its peer nodes in said GUI.

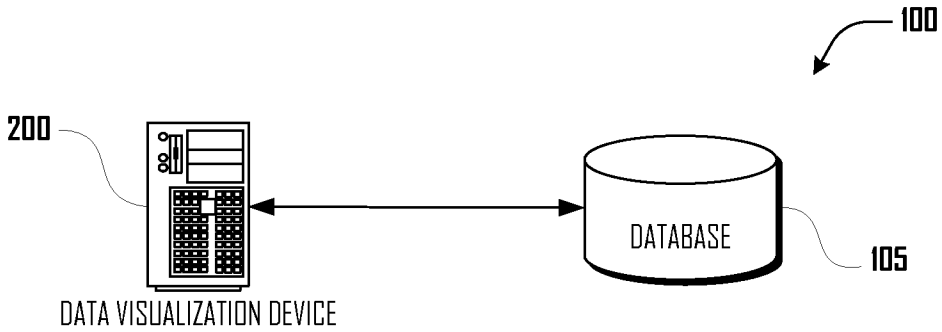

Fig.1

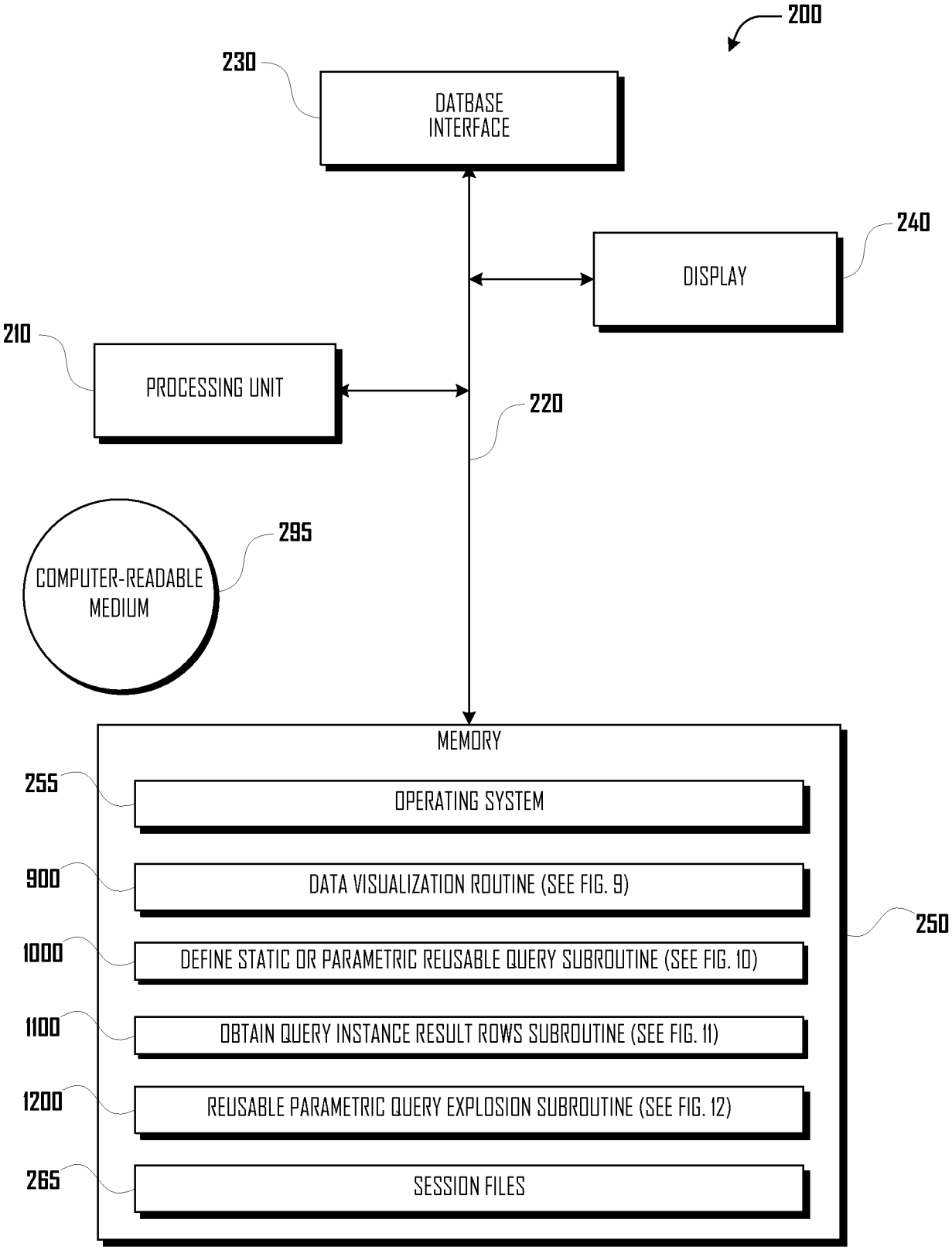
200
230
DATBASE INTERFACE
240
DISPLAY
210
PROCESSING UNIT
220
295
COMPUTER-READABLE MEDIUM
MEMORY
255
OPERATING SYSTEM
900
DATA VISUALIZATION ROUTINE (SEE FIG. 9)
250
1000
DEFINE STATIC OR PARAMETRIC REUSABLE QUERY SUBROUTINE (SEE FIG. 10)
1100
OBTAIN QUERY INSTANCE RESULT ROWS SUBROUTINE (SEE FIG. 11)
1200
REUSABLE PARAMETRIC QUERY EXPLOSION SUBROUTINE (SEE FIG. 12)
265
SESSION FILES

Fig.2

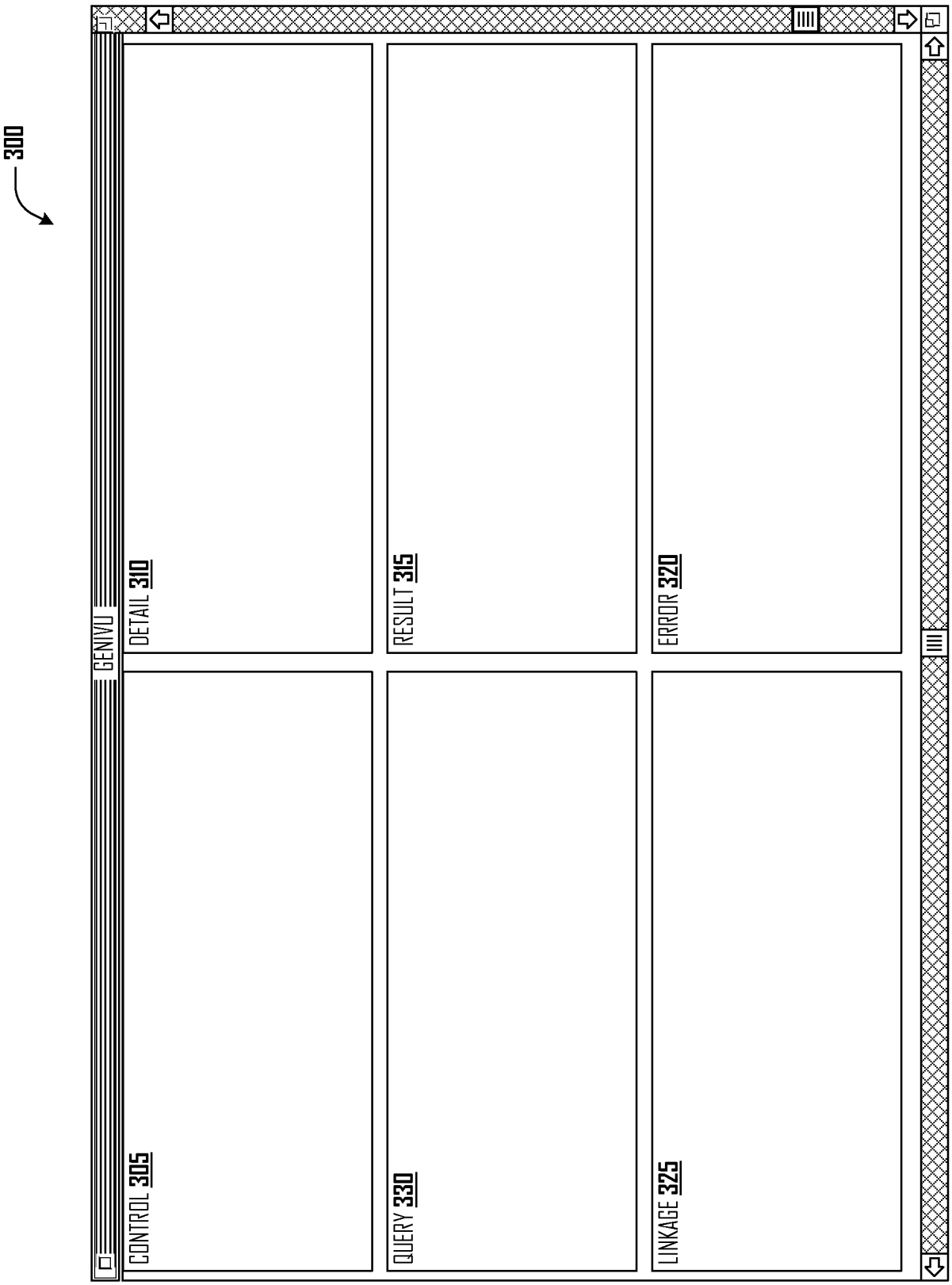
300
GENIVU
DETAIL 310
RESULT 315
ERROR 320
CONTROL 305
QUERY 330
LINKAGE 325

Fig.3

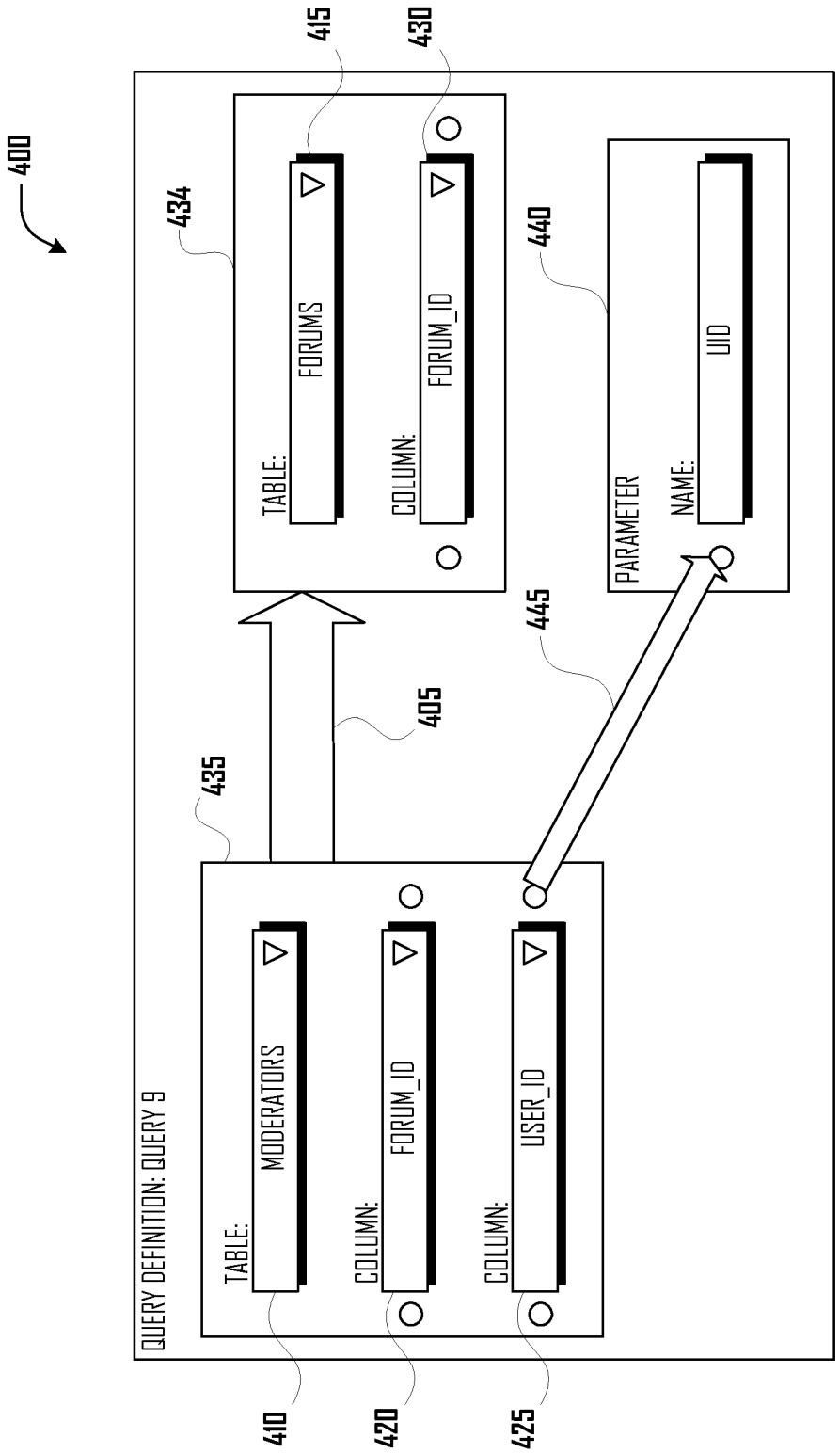
400
415
430
434
440
TABLE:
FORUMS
COLUMN:
FORUM_ID
PARAMETER
NAME:
UID
445
405
435
QUERY DEFINITION: QUERY 9
TABLE:
MODERATORS
COLUMN:
FORUM_ID
COLUMN:
USER_ID
410
420
425

Fig. 4

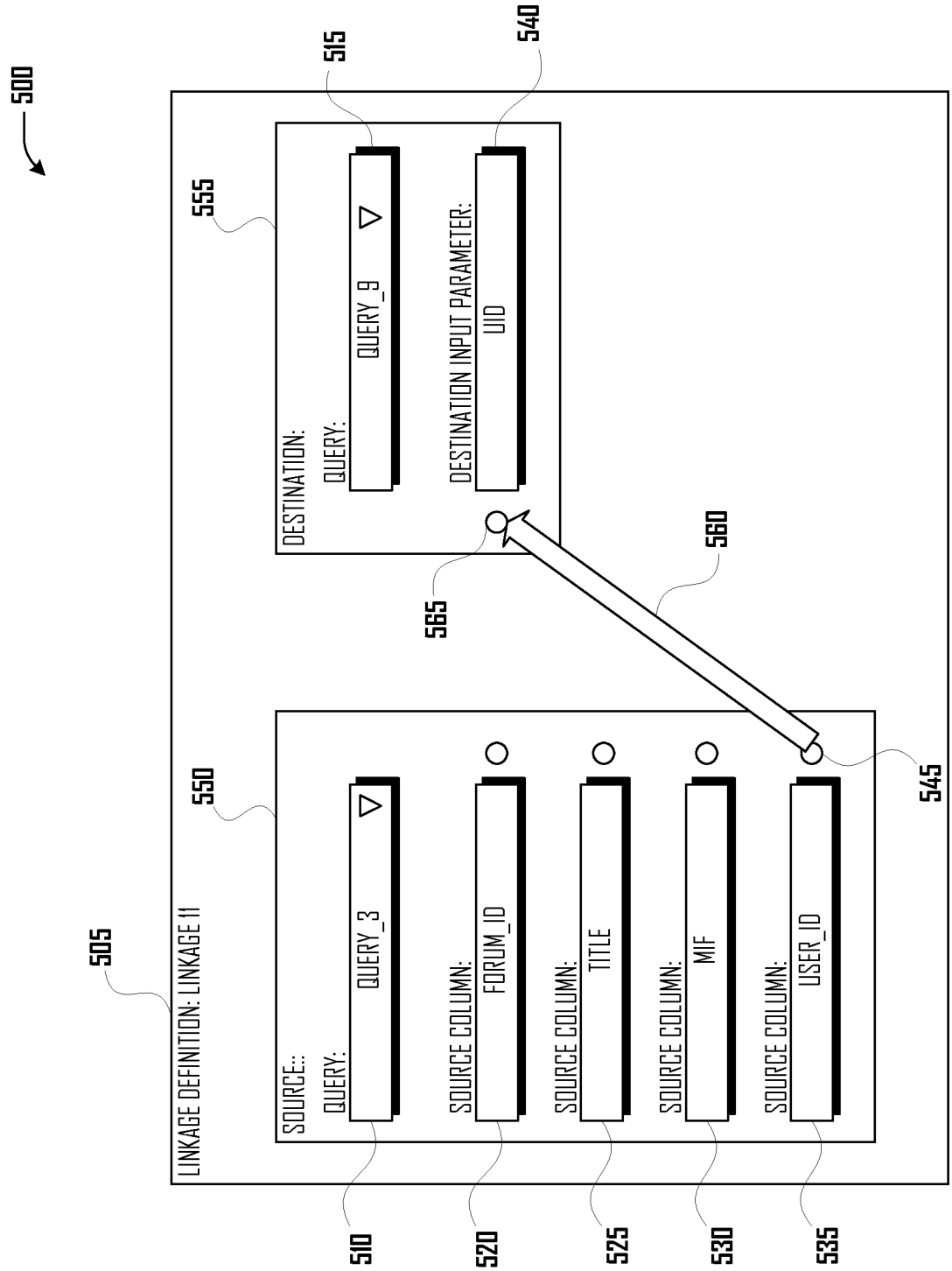
500
505
LINKAGE DEFINITION: LINKAGE 11
550
SOURCE::
QUERY:
QUERY_3
SOURCE COLUMN:
FORUM_ID
SOURCE COLUMN:
TITLE
SOURCE COLUMN:
MIF
SOURCE COLUMN:
USER_ID
555
DESTINATION:
QUERY:
QUERY_9
DESTINATION INPUT PARAMETER:
UID
515
540
560
565
545
510
520
525
530
535

Fig.5

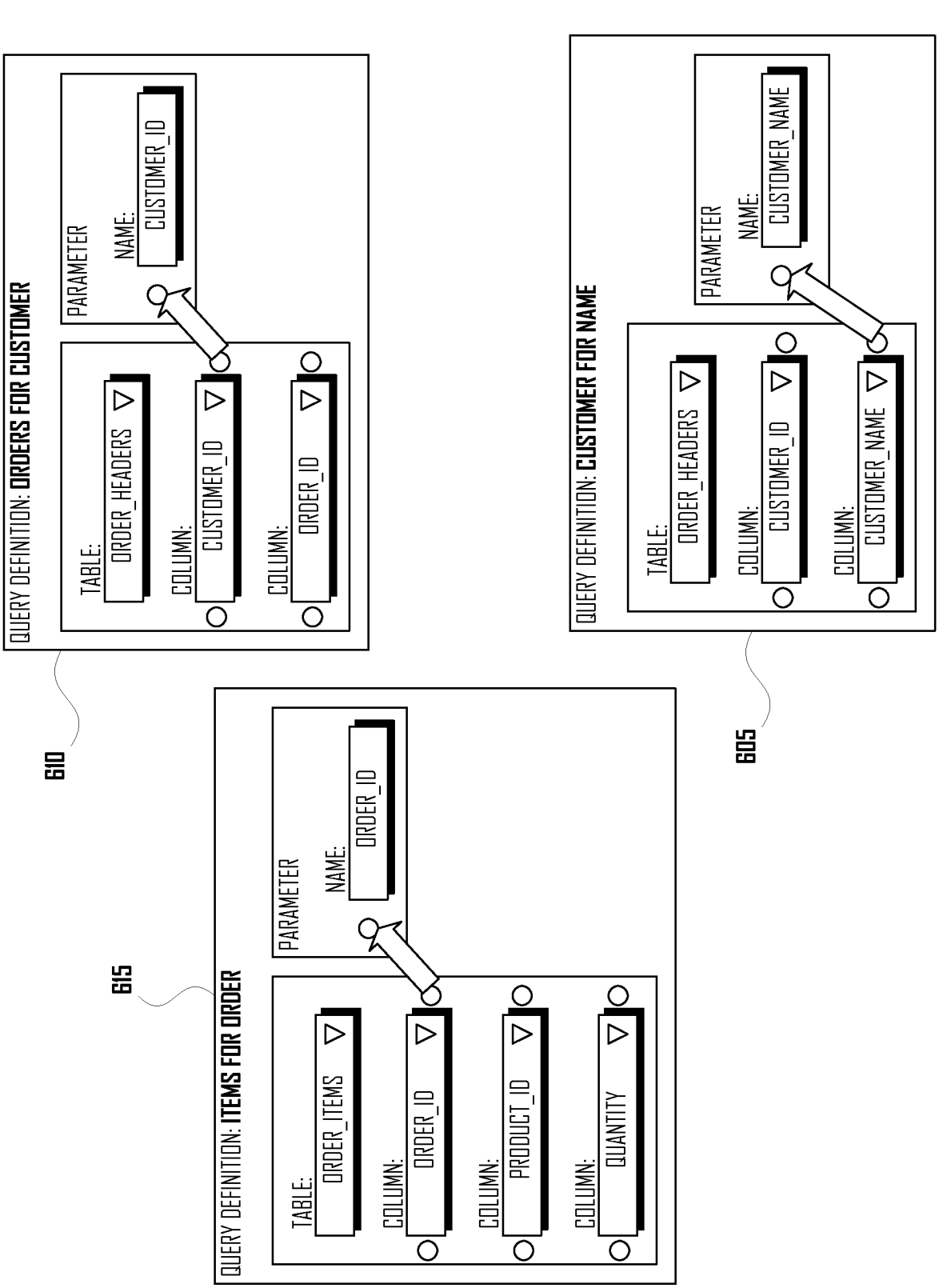
QUERY DEFINITION: ORDERS FOR CUSTOMER
PARAMETER
NAME:
CUSTOMER_ID
TABLE:
ORDER_HEADERS
COLUMN:
CUSTOMER_ID
COLUMN:
ORDER_ID
610
QUERY DEFINITION: CUSTOMER FOR NAME
PARAMETER
NAME:
CUSTOMER_NAME
TABLE:
ORDER_HEADERS
COLUMN:
CUSTOMER_ID
COLUMN:
CUSTOMER_NAME
605
615
QUERY DEFINITION: ITEMS FOR ORDER
PARAMETER
NAME:
ORDER_ID
TABLE:
ORDER_ITEMS
COLUMN:
ORDER_ID
COLUMN:
PRODUCT_ID
COLUMN:
QUANTITY

Fig.6

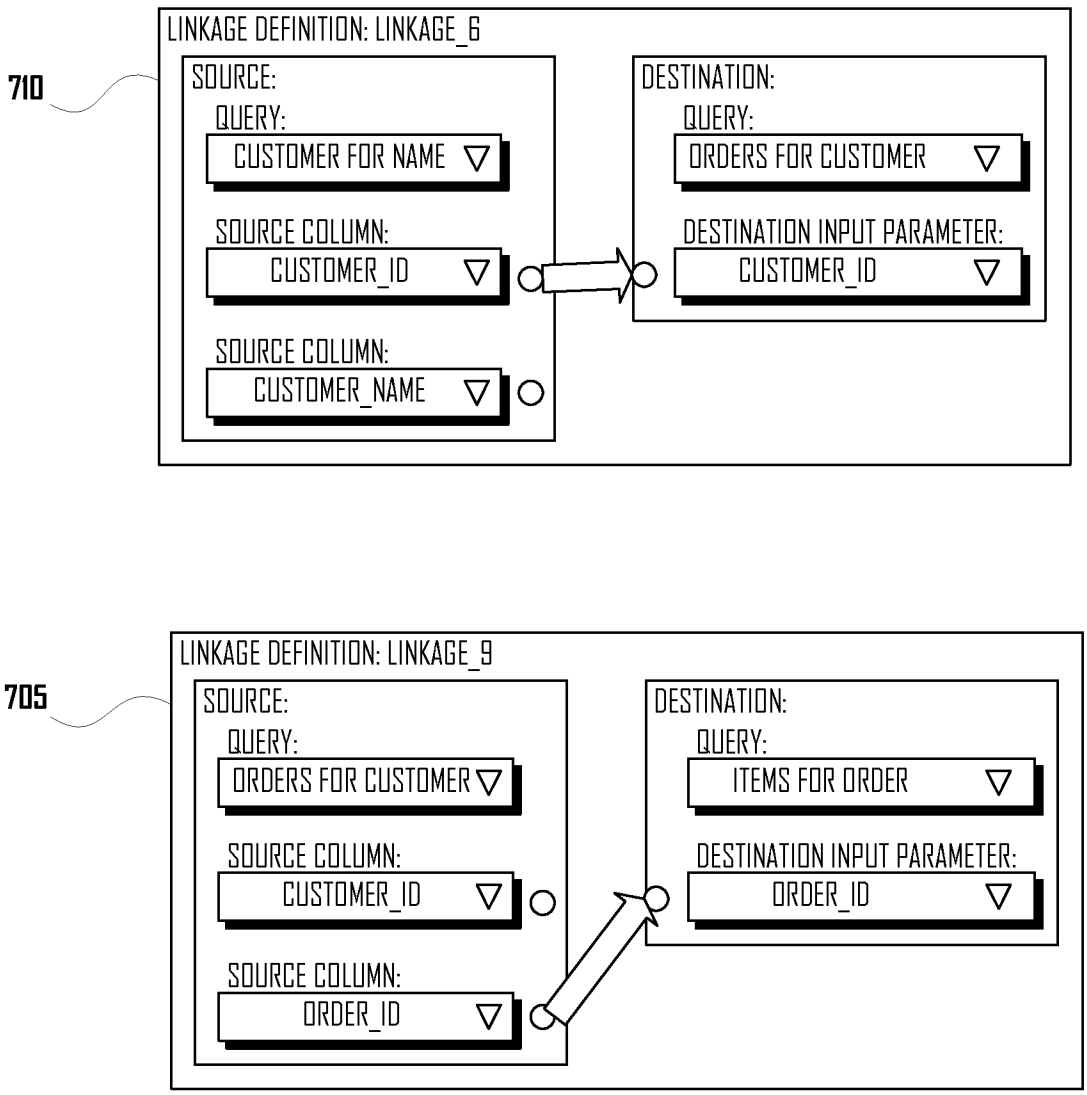
710
LINKAGE DEFINITION: LINKAGE_6
SOURCE:
QUERY:
CUSTOMER FOR NAME
SOURCE COLUMN:
CUSTOMER_ID
SOURCE COLUMN:
CUSTOMER_NAME
DESTINATION:
QUERY:
ORDERS FOR CUSTOMER
DESTINATION INPUT PARAMETER:
CUSTOMER_ID
705
LINKAGE DEFINITION: LINKAGE_9
SOURCE:
QUERY:
ORDERS FOR CUSTOMER
SOURCE COLUMN:
CUSTOMER_ID
SOURCE COLUMN:
ORDER_ID
DESTINATION:
QUERY:
ITEMS FOR ORDER
DESTINATION INPUT PARAMETER:
ORDER_ID

Fig.7

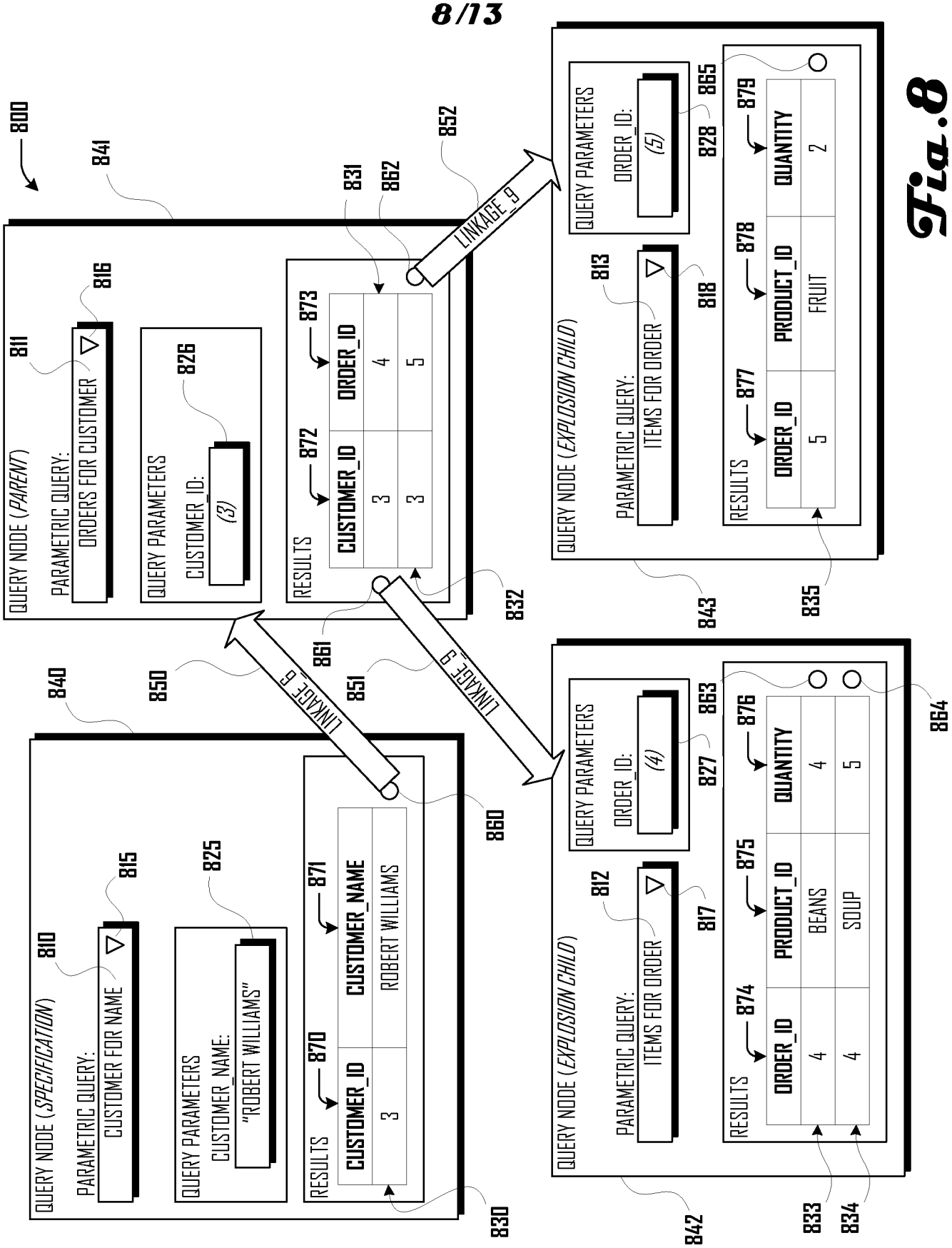
800
841
831
862
852
811
816
QUERY NODE (PARENT)
PARAMETRIC QUERY:
ORDERS FOR CUSTOMER
QUERY PARAMETERS
826
CUSTOMER_ID:
(3)
RESULTS
872
873
CUSTOMER_ID
ORDER_ID
3
4
3
5
LINKAGE_9
832
QUERY NODE (EXPLOSION CHILD)
PARAMETRIC QUERY:
ITEMS FOR ORDER
813
818
QUERY PARAMETERS
ORDER_ID:
(5)
828
865
RESULTS
877
878
879
ORDER_ID
PRODUCT_ID
QUANTITY
5
FRUIT
2
843
835
840
850
861
851
LINKAGE_6
LINKAGE_9
860
QUERY NODE (SPECIFICATION)
810
PARAMETRIC QUERY:
CUSTOMER FOR NAME
815
825
QUERY PARAMETERS
CUSTOMER_NAME:
"ROBERT WILLIAMS"
RESULTS
870
871
CUSTOMER_ID
CUSTOMER_NAME
3
ROBERT WILLIAMS
830
QUERY NODE (EXPLOSION CHILD)
812
817
PARAMETRIC QUERY:
ITEMS FOR ORDER
QUERY PARAMETERS
ORDER_ID:
(4)
827
863
RESULTS
874
875
876
ORDER_ID
PRODUCT_ID
QUANTITY
4
BEANS
4
4
SOUP
5
842
833
834
864

Fig.8

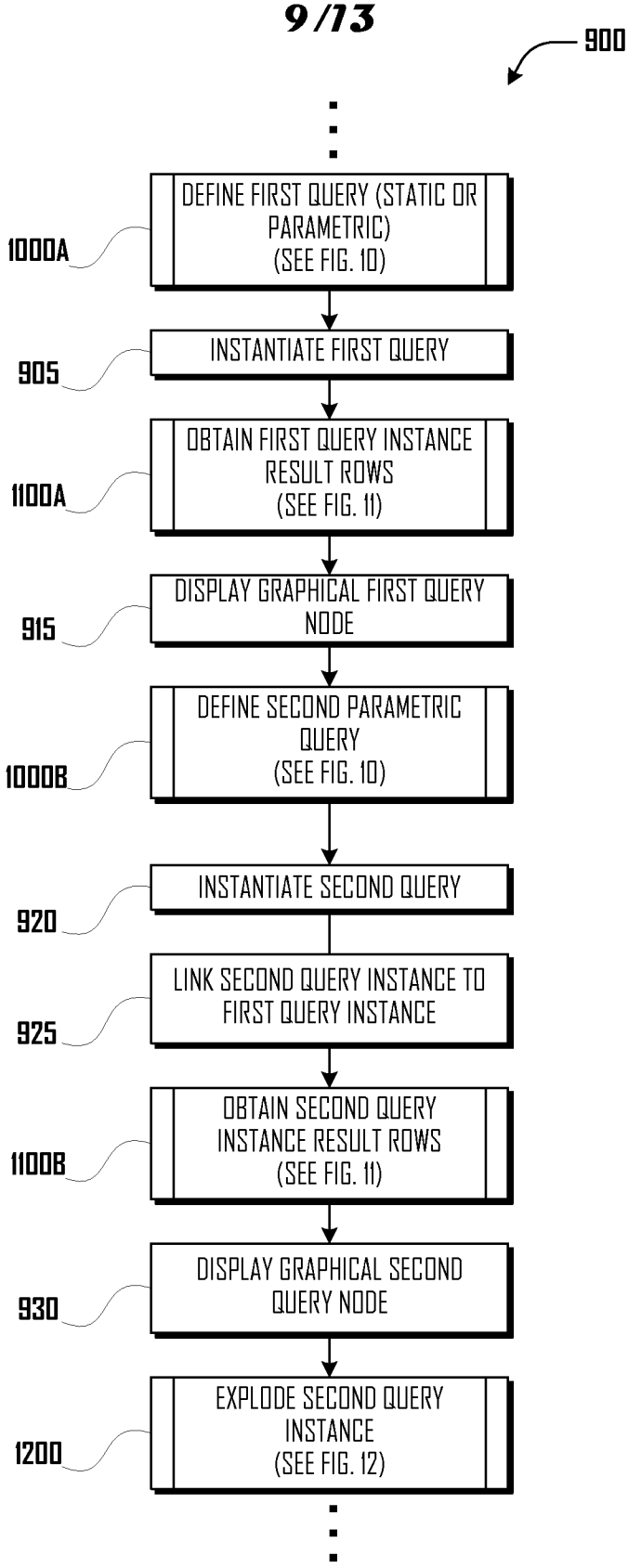
900
DEFINE FIRST QUERY (STATIC OR PARAMETRIC) (SEE FIG. 10)
1000A
INSTANTIATE FIRST QUERY
905
OBTAIN FIRST QUERY INSTANCE RESULT ROWS (SEE FIG. 11)
1100A
DISPLAY GRAPHICAL FIRST QUERY NODE
915
DEFINE SECOND PARAMETRIC QUERY (SEE FIG. 10)
1000B
INSTANTIATE SECOND QUERY
920
LINK SECOND QUERY INSTANCE TO FIRST QUERY INSTANCE
925
OBTAIN SECOND QUERY INSTANCE RESULT ROWS (SEE FIG. 11)
1100B
DISPLAY GRAPHICAL SECOND QUERY NODE
930
EXPLODE SECOND QUERY INSTANCE (SEE FIG. 12)
1200

Fig.9

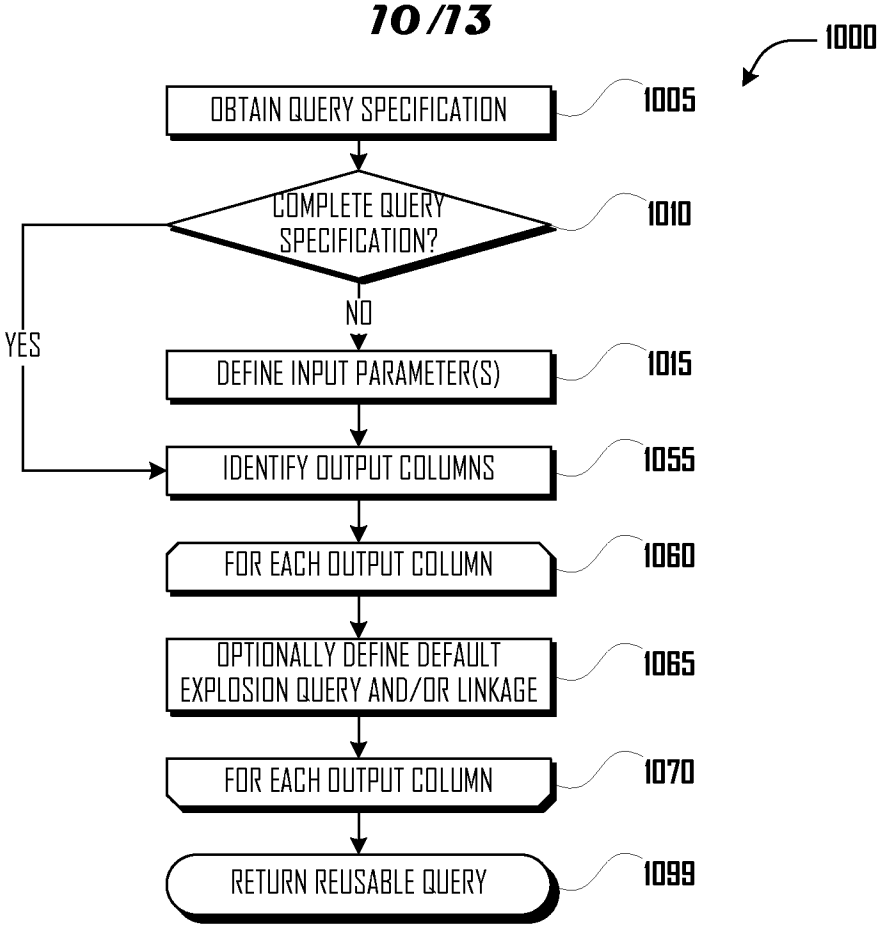
1000
OBTAIN QUERY SPECIFICATION
1005
COMPLETE QUERY SPECIFICATION?
1010
YES
NO
DEFINE INPUT PARAMETER(S)
1015
IDENTIFY OUTPUT COLUMNS
1055
FOR EACH OUTPUT COLUMN
1060
OPTIONALLY DEFINE DEFAULT EXPLOSION QUERY AND/OR LINKAGE
1065
FOR EACH OUTPUT COLUMN
1070
RETURN REUSABLE QUERY
1099

Fig.10

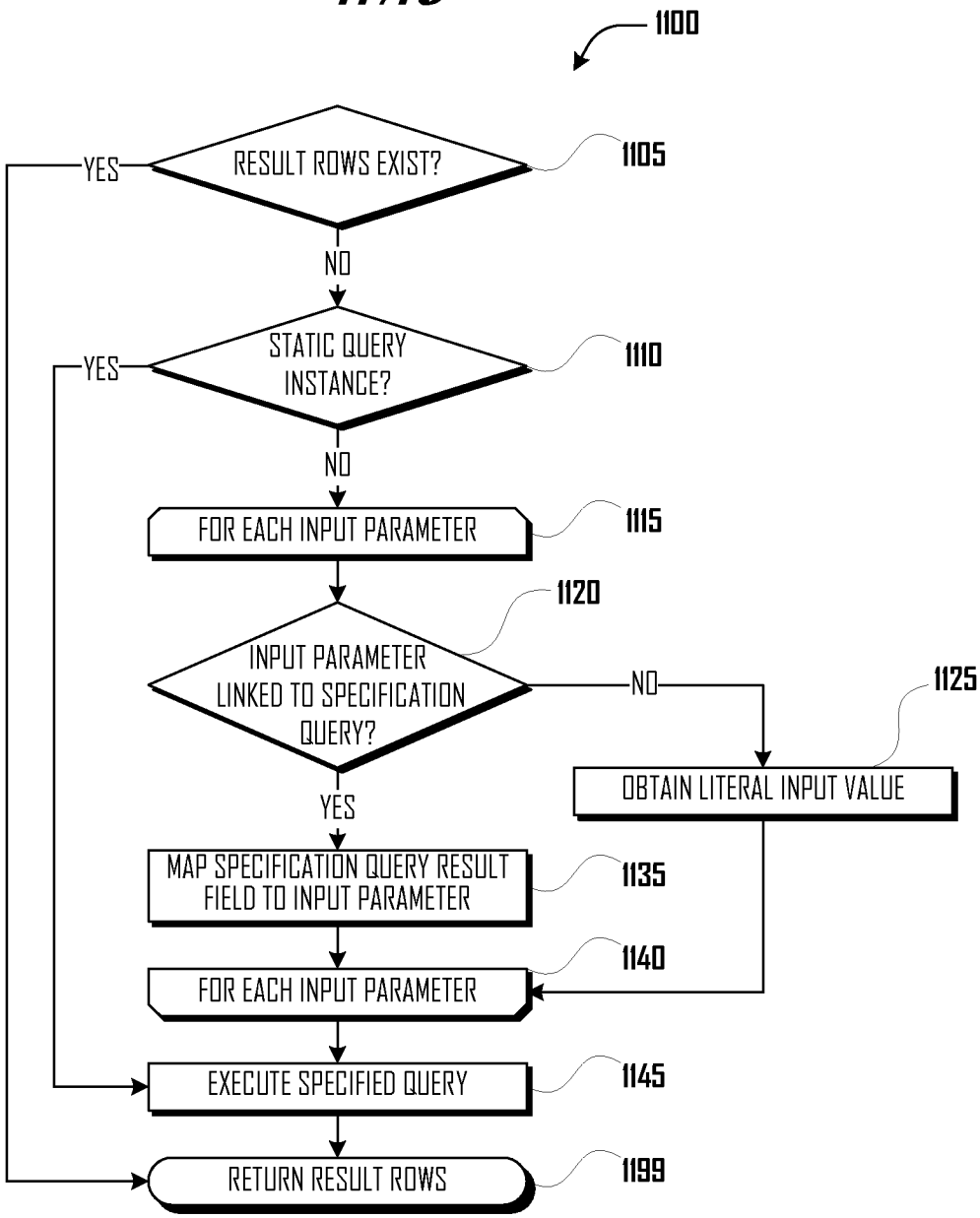
1100
RESULT ROWS EXIST?
1105
YES
NO
STATIC QUERY INSTANCE?
1110
YES
NO
FOR EACH INPUT PARAMETER
1115
1120
INPUT PARAMETER LINKED TO SPECIFICATION QUERY?
NO
1125
OBTAIN LITERAL INPUT VALUE
YES
MAP SPECIFICATION QUERY RESULT FIELD TO INPUT PARAMETER
1135
1140
FOR EACH INPUT PARAMETER
EXECUTE SPECIFIED QUERY
1145
RETURN RESULT ROWS
1199

Fig.11

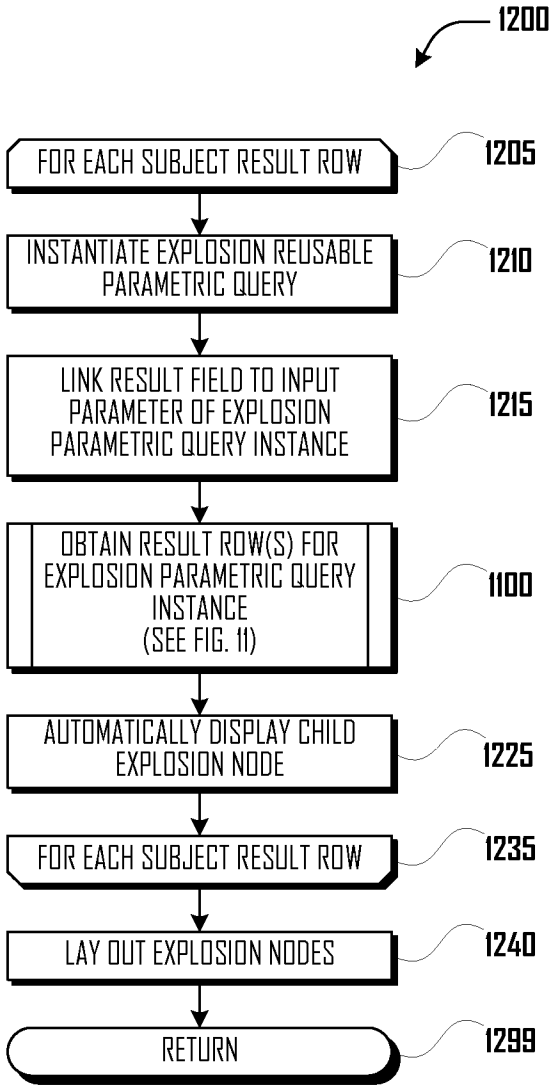
1200
FOR EACH SUBJECT RESULT ROW
1205
INSTANTIATE EXPLOSION REUSABLE PARAMETRIC QUERY
1210
LINK RESULT FIELD TO INPUT PARAMETER OF EXPLOSION PARAMETRIC QUERY INSTANCE
1215
OBTAIN RESULT ROW(S) FOR EXPLOSION PARAMETRIC QUERY INSTANCE (SEE FIG. 11)
1100
AUTOMATICALLY DISPLAY CHILD EXPLOSION NODE
1225
FOR EACH SUBJECT RESULT ROW
1235
LAY OUT EXPLOSION NODES
1240
RETURN
1299

Fig.12

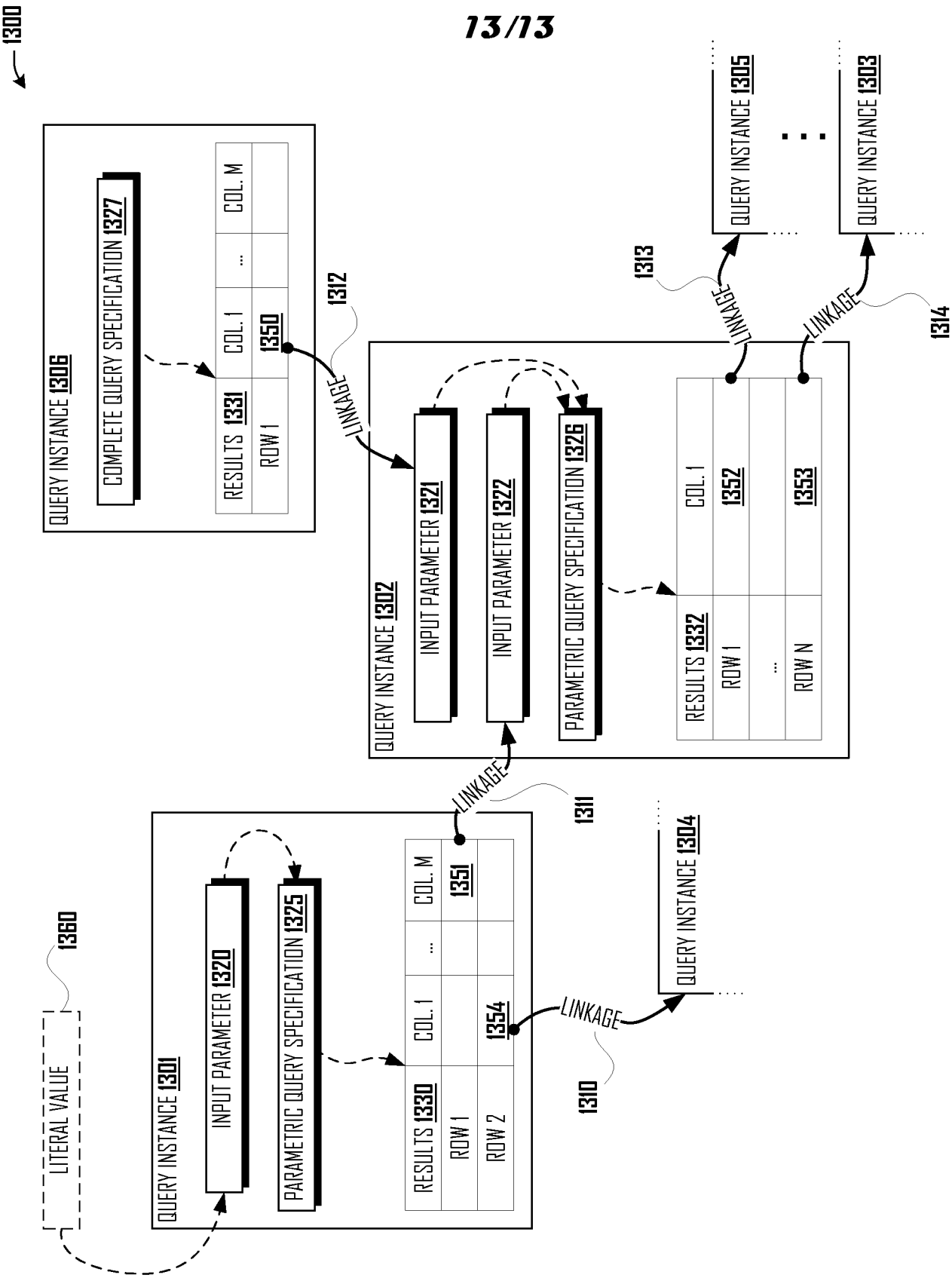
1300
QUERY INSTANCE 1306
COMPLETE QUERY SPECIFICATION 1327
RESULTS 1331
ROW 1
COL. 1
1350
...
COL. M
1312
LINKAGE
QUERY INSTANCE 1302
INPUT PARAMETER 1321
INPUT PARAMETER 1322
PARAMETRIC QUERY SPECIFICATION 1326
RESULTS 1332
ROW 1
1352
...
ROW N
1353
COL. 1
1313
LINKAGE
1314
LINKAGE
QUERY INSTANCE 1305
QUERY INSTANCE 1303
1311
LINKAGE
QUERY INSTANCE 1301
INPUT PARAMETER 1320
PARAMETRIC QUERY SPECIFICATION 1325
RESULTS 1330
ROW 1
ROW 2
1354
COL. 1
...
COL. M
1351
1360
LITERAL VALUE
1310
LINKAGE
QUERY INSTANCE 1304

Fig.13