

GRAPHICAL DEVELOPMENT METHOD, MEDIUM, DEVICE, AND APPARATUS

5 **[0001]** This disclosure claims the benefits of priority to Chinese Patent Application No. 201910299903.7 and entitled “GRAPHICAL DEVELOPMENT METHOD, MEDIUM, DEVICE, AND APPARATUS” filed April 15, 2019, which is incorporated herein by reference in its entirety.

TECHNICAL FIELD

[0002] The present disclosure relates to the technical field of computer program development, and more particularly to graphical development method, medium, device, and apparatus.

BACKGROUND

10 **[0003]** Structured Query Language, SQL, is a database query and program designing language used to access data, and to query, update and manage relational database systems. It is also an extension of the database script file.

15 **[0004]** In prior art, when developing a program using SQL statements, a way of development is editing SQL scripts in text or writing SQL statements.

[0005] By the existing development approach, developers must be very familiar a specific syntax of SQL. For a developer who is exposed to the SQL language for the first time, learning and familiarizing with a new SQL syntax involves certain learning and time costs. Thus, development efficiency is compromised.

SUMMARY

20 **[0006]** The technical problem to be solved by the present disclosure is that editing an SQL statement by means of a command line or a text involves mastering a grammatical content, and compromises the efficiency of program development.

25 **[0007]** The present disclosure solves the above-mentioned technical problem, and an embodiment of the present disclosure provides a graphical development method, comprising:

[0008] encapsulating, in each DAG component, an operator for implementing a function of the component, and a name of another DAG component allowed to be connected;

[0009] presenting a plurality of visualized DAG components to a user;

[0010] receiving selection of a plurality of DAG components by the user, and establishing a connectivity of the DAG components according to the function and the component allowed to be connected of the respective DAG component, thereby generating a DAG component graph.

[0011] Preferably, the encapsulating further comprises: encapsulating one or more parameter or attribute configuration item;

[0012] receiving selection of the plurality of DAG component by the user further comprises:

[0013] displaying a parameter or attribute configuration panel for the DAG component selected by the user;

[0014] receiving the configuration parameter or attribute data input by the user, and verifying the data in real time.

[0015] Preferably, the method further comprising:

[0016] re-verifying, each time when the connectivity is changed by adding a DAG component into the connectivity, a downstream path of the added DAG component, wherein the downstream path includes the respective DAG components and the links formed thereby in the connectivity;

[0017] re-verifying, when any of the DAG components is clicked, links for the DAG components in a downstream path of the clicked DAG component.

[0018] Preferably, the re-verifying comprises: checking correctness of a field list and type of one or more input/output of each DAG component.

[0019] Preferably, the method further comprising: selecting a region including a plurality of DAG components in the established connectivity, and saving the region as a callable template.

[0020] Preferably, the method further comprising: analyzing code relationship of the DAG component graph after generating the DAG component graph.

[0021] Preferably, the analyzing comprises:

[0022] resolving the connectivity formed by the respective DAG components and links into a data structure;

[0023] decomposing the data structure into a plurality of logical operators according to a statement configuration;

[0024] generating statement codes correspondingly to the plurality of logical operators.

[0025] Preferably, the method further comprising: receiving a switch instruction and performing an operation inverse to the analyzing to form the DAG component graph, after generating the statement codes.

[0026] An embodiment of the present disclosure further provides a computer-readable storage medium, wherein the computer-readable storage medium stores a computer program which, when being executed by a processor, implements the method as above.

[0027] An embodiment of the present disclosure further provides a computer device incorporated with the above computer storage medium.

[0028] An embodiment of the present disclosure further provides a graphical development apparatus comprising:

5 **[0029]** an encapsulation module configured to encapsulate, in each DAG component, an operator for implementing a function of a component, and a name of another DAG component allowed to be connected;

[0030] a presentation module configured to present a plurality of visualized DAG components to a user;

10 **[0031]** a graphics module configured to receive selection of a plurality of DAG components by the user, establish a connectivity of the DAG components according to the function and the component allowed to be connected of the respective DAG component, thereby generate a DAG component graph.

[0032] Preferably, the encapsulating further comprises: encapsulating one or more parameter or
15 attribute configuration item;

[0033] the graphics module is coupled to a parameter verification module, which is configured to display a parameter or attribute configuration panel for the DAG component selected by the user,

[0034] receive the configuration parameter or attribute data input by the user and verify the data in
20 real time.;

[0035] Preferably, the graphics module is further coupled to a re-verification module, which is configured to re-verify, each time when the connectivity is changed by adding a DAG component into the connectivity, a downstream path of the added DAG component, wherein the downstream path includes the respective DAG components and the links formed thereby in the
25 connectivity;

[0036] or, re-verify, when any of the DAG components is clicked, links for the DAG components in a downstream path of the clicked DAG component.;

[0037] Preferably, the graphics module is further coupled to a template module, which is configured to select a region including a plurality of DAG components in the established
30 connectivity, and save it as a callable template.

[0038] Preferably, the graphics module is further coupled to an analysis module, which comprises:

[0039] a first sub-module configured to resolve the connectivity formed by the respective DAG components and links into a data structure;

[0040] a second sub-module configured to decompose the data structure into a plurality of logical

operators according to a statement configuration; and

[0041] a third sub-module configured to generate statement codes correspondingly to the plurality of logical operators.

[0042] Preferably, the analysis module is coupled to a conversion module, which is configured to receive a switch instruction and perform an operation inverse to the analysis module to form the DAG component graph.

[0043] Through the above-mentioned embodiments, an operator for implementing a function of a corresponding components by code is encapsulated into a DAG component, and the software development process is displayed in a graphical manner, which simplifies the development process of a user. Thus, the user does not pay more attention to the expression of each specific instruction and the grammatical relationship. Such a development mode abstracts and encapsulates the operators of the code into visual and configurable front-end components, which is more intuitive. It is easy for the user to understand that the logical association between respective DAG components is abstracted as a link, and the user only needs to understand the processing capability of the DAG component, and can complete program task development by dragging and configuring the component. Even new users with no language background need not care about the underlying grammar.

BRIEF DESCRIPTION OF DRAWINGS

[0044] Figure 1 is a method flow graph of an embodiment of the present disclosure;

[0045] Figure 2 is an interface graph illustrating parameter configuration in an embodiment of the present disclosure;

[0046] Figure 3 is a graph of a DAG assembly in an embodiment of the present disclosure;

[0047] Figure 4 is a schematic graph of a re-verifying process according to an embodiment of the present disclosure;

[0048] Figure 5 is a graph illustrating a layer structure transformation of a DAG component and a SQL code according to an embodiment of the present disclosure;

[0049] Figure 6 is a graph of another DAG assembly according to an embodiment of the present disclosure;

[0050] Figure 7 is three resolved sub-graphs of an embodiment of the present disclosure;

[0051] Figure 8 is a block graph of an apparatus structure according to an embodiment of the present disclosure.

DESCRIPTION OF EMBODIMENTS

[0052] The principles and features of the present disclosure are described below with reference to the accompanying drawings, which are intended to illustrate rather than to limit the scope of the disclosure.

5 **[0053]** The following describes the embodiments of the present disclosure in detail with reference to the accompanying drawings. Referring to Figure 1, an embodiments of the present disclosure comprises:

[0054] S11: Encapsulate, in each DAG component, an operator for implementing a function of the component, and a name of another DAG component allowed to be connected;

10 **[0055]** S12: Present a plurality of visualized DAG components to a user;

[0056] S13: Receive selection of a plurality of DAG components by the user, and establish, by the user, a connectivity of the DAG components according to the function and the component allowed to be connected of the respective DAG component, thereby generating a DAG component graph. The user may select one or more DAG components by dragging.

15 **[0057]** Through the above-mentioned embodiments, an operator for implementing a function of a corresponding components by code is encapsulated into a DAG component, and the software development process is displayed in a graphical manner, which simplifies the development process of a user. Thus, the user does not pay more attention to the expression of each specific instruction and the grammatical relationship. Such a development mode abstracts and
20 encapsulates the operators of the code into visual and configurable front-end components, which is more intuitive. The logical association between respective DAG components is abstracted as a link, and the user only needs to understand the processing capability of the DAG component, and can complete program task development by dragging and configuring the component. Even new users need not care about the underlying grammar.

25 **[0058]** The DAG development mode is more consistent with thinking habits of a user. In development by the DAG mode, the user can implement the process of expressing a program task according to the thinking habit, the sequence of components and the relationship between them. An example related to traffic logic needs to be listed. The DAG component graph actually also reflects the data processing flow. For a complex data processing process, it can also make
30 its expression process more intuitive through the dragging and configuration editing. At the same time, the editing mode based on DAG can also effectively reduces the cost of subsequent development and maintenance. A software process framework can be built more quickly, which effectively improves the efficiency.

[0059] The method of the present disclosure can be applied to development programs in multiple

languages, such as SQL language, Scala language, etc. Each language has an operator supported respectively, and the operator can be described by code. One or more operators can be encapsulated to realize the functions of a component. The way of flow calculation and graph calculation can be implemented according to this scheme. In an embodiment of the present disclosure, the SQL language is exemplified.

[0060] Preferably, in the above embodiment, the encapsulating further comprises: encapsulating one or more parameter or attribute configuration item;

[0061] The process of receiving selection of a plurality of DAG components by the user further comprises:

[0062] displaying a parameter or attribute configuration panel for the DAG component selected by the user;

[0063] receiving the configuration parameter or attribute data input by the user, and verifying the data in real time.

[0064] The function of each DAG component is provided with a configurable parameter range, or attribute configuration range, etc. If the value configured by the user exceeds the range value, a problem pop-up window will be displayed to the user after comparing the values. Compared with the prior art where a situation of configuration error can only be seen in the later compilation process that requires the coding to be already completed, As shown in Figure 2, in the software development process, this embodiment can find the content of configuration error and improve the accuracy rate of code.

[0065] Preferably, the process also includes:

[0066] re-verifying, each time when the connectivity is changed by adding a DAG component into the connectivity, a downstream path of the added DAG component, wherein the downstream path includes the respective DAG components and the links formed thereby in the connectivity. As shown in Figure 3, on the basis of the TT component, the user drags the component GroupBy and uses a link to connect; after the connection, the user will timely detect whether the path formed by the TT before the GroupBy component node is correct, which can be verified by means of reckoning of schema. The connection formed by the TT component and the GroupBy component can also be a sub-graph, which forms a single SQL statement.

[0067] Through the above-mentioned real-time checking, the DAG visualization component normalizes the usage mode of the operator, such as what components can be connected in the upstream and downstream of the component, what content is allowed to be configured for the attribute configuration item, and what fields can be used, etc., all of these being strictly normalized, reducing the possibility of making error by the processing logic. The user is able to

be prompted in real time even if he has made an error in configuration. This facilitates the user to detect in real time whether the connected components and connectivity are correct during each DAG component connection.

[0068] Preferably, a user can also click a certain node, for example, the user clicks one of the select nodes, and the system will re-verify the path before the node, TT-GroupBy-filter-select, take this path and node as a task, re-verify whether the formed path and node are correct, and verify the correctness of this path and node by means of reckoning of schema. Thus, the debugging capability of the implementation is enhanced.

[0069] Preferably, in the process of real-time re-verification using schema, when there is a connection action between components, or when the components are configured or modified, the following process is are triggered: (see Figure 4), which process comprises:

[0070] S21: Check correctness of links of each component from top to bottom;

[0071] 211 Check the types of components allowed to be connected downstream of each component (belonging to the component's implicit configuration);

[0072] 212 Checking fails, indicate wrong links, and terminate current checking;

[0073] S22, Check correctness of each component parameter configuration;

[0074] 221 Check the component configuration parameters for compliance with parameter rules, such as whether is a parameter type (shaping, string), whether null is allowed, etc.;

[0075] 222 Check if the configuration items and the fields referenced in the configuration expression come from the output fields of the upstream component. For example, an upstream source table component outputs two fields aa, bb, and no fields with names other than aa, bb can be referenced in the expression of a downstream Select component configuration;

[0076] S23, Calculate the output schema of each component;

[0077] 231 Perform reckoning of the output shcema when the link and parameter configuration check are passed;

[0078] 232 Schema mainly includes an output field list of components and a field type (e.g.: aa String, bb String);

[0079] 233 The schema output by an upstream component is the input schema of a downstream component;

[0080] 234 Each component has a fixed shcema reckoning logic, and the parsing method of different components may vary. For example, a Select component obtains the output field by parsing a parameter expression: The upstream input schema is a String, b String, the selectFields of the Select component is configured as aa as a, max (bb) as b, and the schema to the output is parsed by expression parsing (using SQL parsing tools):aa String, bb String; For a Join

component: assuming that in the two upstream inputs, the schema of one output is aa String, bb String, and the other input is cc String, dd String, thus the input schema of the Join component is the union of the two upstream schema: aa String, bb String, cc String, dd String. Through such re-verification, it can be found whether the logic of the entire SQL is correct. When the checking of each DAG component node fails, the checking of the DAG component graph is terminated and a node configuration error message is output. The error correction capability of the code is thereby enhanced.

[0081] Preferably, in order to facilitate the user in further developing the software more conveniently and to improve efficiency, in the established connectivity formed by a plurality of DAG components, a region including a plurality of partial DAG components which have formed the connectivity can be selected, and the plurality of DAG components selected by region can be saved as a template. When making a subsequent editing again, the template can be directly dragged and added to the whole software under development. The contents of the template include some common configuration attributes of components (such as connection information of a table, field information of a table, expression information, etc.) and links between components, etc. Therefore, the code can be reused quickly, and the development efficiency of software is improved.

[0082] Preferably, further comprising analyzing the component relationship graph formed by the DAG to form code that is ultimately used for compilation. The analysis process described above resolved the components of the DAG component graph into final code. For example, the DAG component of SQL is finally resolved into an executable SQL code. An example of a specific resolve process is described in detail below.

[0083] Referring to FIGURE 5, in an embodiment, a DAG component that primarily includes a rendered front-end that forms a connectivity is located in the first row of the graph.

[0084] During resolving, each component of the DAG of the first row is resolved downward. Each DAG component of the first row, which forms a view of the connectivity as shown in Figure 6. These DAG component graphs, after being resolved, form a node list and an edge list, and constitute one json data body, json being a subset of the syntax of the JavaScript object representation. Each node stores a component type and a component parameter configured by a user in the component configuration panel, and the edge data stores a directed connectivity between components. See the view layer of the second row in Figure 4.

[0085] After receiving the DAG data volume, the view layer of the second row decomposes the DAG sub-graph into a plurality of sub-graphs, and the purpose of decomposing is to make each sub-graph eventually generate a separate SQL statement; and the basis of decomposing is to

judge whether the DAG component of each node needs to be disconnected at this point (for example, the create view node of the present example needs to be disconnected from the subsequent nodes so as to form one independent SQL expression statement). After decomposing, the contents of the DAG abstraction component layer, i.e. the abstract component layer, of the third row as shown in Figure 5 are formed.

[0086] The DAG abstraction component layer decomposes the DAG component into three sub-graphs, and the three sub-graphs after decomposing are shown in Figure 6. This corresponds to what is shown in the three dashed boxes in the third row of Figure 5. In the above-mentioned transmission process of each layer, the data structure of each layer content is also adjusted accordingly.

[0087] After the abstract component layer is connected to the data of the view layer, the content is further resolved to obtain the content of the basic logical operator layer, as shown in the fourth row of Figure 5. Decomposing the DAG component into one or more underlying SQL operator structures (depending on the functionality of the DAG component encapsulation). For example, the GroupBy component of Figure 5 or Figure 6 may be decomposed into a GroupBy operator and a Select operator.

[0088] The basic logical operator layer, successively traversing the logical operator sub-graphs parsed and decompose from the abstract component layer, generates corresponding SQL codes.

[0089] For example, a GroupBy component can be decomposed into a GroupBy operator and a Select operator, and the generated statement is `CREATE VIEW view_4cc AS SELECT aa ,max(bb)AS bb FROM AA GROUP BY aa ;`

[0090] The subgraph corresponding to the GroupBy component is subgraph 1 on the left of Figure 7. In the middle is subgraph 2 and on the right is subgraph 4. In the basic logical operator layer in Figure 5, the component in the middle corresponds to subgraph 2 in Figure 7, and the component on the right corresponds to subgraph 3 in Figure 7.

[0091] The basic logical operator layer, the SQL generated by subgraph 2 and the SQL generated by subgraph 3 are made respectively as follows:

[0092] The SQL statement code generated by subgraph 2:

[0093] `SELECT aa , bb FROM view_4cc WHERE aa > 0 and aa < 100;`

[0094] The SQL statement code generated in subgraph 3:

[0095] `SELECT aa, bb FROM view_4cc WHERE aa > 100`

[0096] Through the above steps, the DAG graph is coded, which facilitates the engineers to further debug and compile the code.

[0097] In the above process, if the user needs to switch from the code state to the view mode of

the DAG component, the window of the code line can be switched to the DAG view mode through the switch button, and at this time, converted to the DAG view according to a reverse operation of the above code analysis process. Taking SQL language as an example, the specific process is as follows:

5 **[0098]** The action of switching from the SQL to DAG triggers the following process:

[0099] The basic logic operator layer receiving the SQL text content edited by the SQL layer, using an SQL parsing tool to parse the SQL text content into an AST syntax tree, traversing and parsing the AST syntax tree to generate a defined basic operator DAG data structure. Wherein, traversing the AST syntax tree can use existing SQL tools to parse each individual SQL statement into an independent logical operator DAG structure according to the final encapsulated function;

10

[00100] The abstract component layer encapsulates one or more logical operators into abstract components according to the abstract component encapsulation requirements, and finally outputs an abstract component DAG data structure; As in this example: Based on the usual grammatical features, groupBy operators must be used in conjunction with select operators because they can be combined into GroupBy abstract components. In this example, three independent logic component DAG structures are finally output, as shown in the upper dashed box of Figure 5.

15

[00101] The DAG view layer combines three independent logical component DAG structures into one independent DAG structure. The principle of combining is to find the input-output associations between subgraphs. For example, in Figures 6 and 7, the CreateView node finally outputs one view view with the name of view_4cc, and the first nodes of subgraphs 2 and 3 read data from the source with the name of view_4cc, and thus it can be determined that subgraphs 2 and 3 have data association with subgraph 1, which are downstream subgraphs of subgraph 1. Combining the three subgraphs forms a complete DAG data structure, such as the graph structure in the complete DAG view layer in Figure 5. Encapsulated components may be encapsulated according to different component functions, such as according to data processing or data queries classification, according to definitions of databases/object encapsulation of computational engines, or administrative authority of databases.

20

25

[00102] The front end, receives the complete DAG graph data structure, and renders it into a visualized DAG edit graph, forming the form as shown in Figure 6.

30

[00103] Preferably, embodiments of the present disclosure provide a computer-readable storage medium in which the steps of the above-described embodiments may be stored in the form of program code which. When the program executed by a processor, the method of the

above-described embodiments is performed.

[00104] Preferably, embodiments of the present disclosure provide a computer apparatus incorporating the computer storage medium described above. Such as providing a server for downloading the code software, or a computer for reading from the medium.

5 **[00105]** Preferably, embodiments of the present disclosure also provide a graphical development apparatus, comprising:

[00106] an encapsulation module configured to encapsulate, in each DAG component, an operator for implementing a function of a component, and a name of another DAG component allowed to be connected;

10 **[00107]** a presentation module configured to present a plurality of visualized DAG components to a user;

[00108] a graphics module configured to receive selection of a plurality of DAG components by the user, establish a connectivity of the DAG components according to the function and the component allowed to be connected of the respective DAG component, thereby generate a DAG
15 component graph.

[00109] Preferably, the encapsulating further comprises: encapsulating one or more parameter or attribute configuration item;

[00110] The graphics module is coupled to a parameter verification module, which is configured to display a parameter or attribute configuration panel for the DAG component selected by the user,
20 and

[00111] Receive the configuration parameter or attribute data input by the user and verify the data in real time.

[00112] Preferably, the graphics module is further coupled to a re-verification module, which is configured to re-verify, each time when the connectivity is changed by adding a DAG
25 component into the connectivity, a downstream path of the added DAG component, wherein the downstream path includes the respective DAG components and the links formed thereby in the connectivity; or,

[00113] re-verify, when any of the DAG components is clicked, links for the DAG components in a downstream path of the clicked DAG component.

30 **[00114]** Preferably, the graphics module is further coupled to a template module which is configured to select a region including a plurality of DAG components in the established connectivity, and save it as a callable template.

[00115] Preferably, the graphics module is further connected with an analysis module, including:

[00116] a first sub-module configured to resolve the connectivity formed by the respective DAG

components and links into a data structure;

[00117] a second sub-module configured to decompose the data structure into a plurality of logical operators according to a statement configuration; and

[00118] a third sub-module configured to generate statement codes correspondingly to the plurality of logical operators.

[00119] Preferably, analysis module is coupled to a conversion module, which is configured to receive a switch instruction and perform an operation inverse to the analysis module to form the DAG component graph.

[00120] Figure. 8 is a block graph of an apparatus structure according to an embodiment of the present disclosure. The apparatus in the embodiments described above may be integrated in various computer devices, or storage media, to implement the method steps in the embodiments.

[00121] Persons skilled in the art may appreciate that the elements and algorithm steps described in connection with the embodiments disclosed herein may be implemented as electronic hardware, computer software, or combinations of both, and that the components and steps of the examples have been described above generally in terms of their functionality in order to clearly illustrate this interchangeability of hardware and software. Whether such functionality is implemented as hardware or software depends upon the particular application and design constraints imposed on the technical solution. Persons skilled in the art may implement the described functionality in varying ways for each particular application, but such implementation decisions should not be interpreted as causing a departure from the scope of the present disclosure.

[00122] It will be clear to a person skilled in the art that, for convenience and brevity of description, reference may be made to corresponding processes in the above-described method embodiments for the specific workings of the systems, devices and units described above and will not be described in detail herein.

[00123] In addition, each functional unit in various embodiments of the present disclosure may be integrated in one processing unit, may exist physically in a single unit, or may have two or more units integrated in one unit. The integrated units described above can be implemented either in the form of hardware or in the form of software functional units.

[00124] The integrated units, if implemented as software functional units and sold or used as stand-alone products, may be stored in a computer readable storage medium. Based on such an understanding, the technical solution of the present disclosure can be substantially or partly contributed to the prior art, or all or part of the technical solution can be embodied in the form of a software product stored in a storage medium, comprising instructions to enable a computer device (which may be a personal computer, a server), or a network device or the like).The

aforementioned storage medium includes: Various media can store program codes, such as USB disks, removable hard disks, Read-Only Memory (ROM), Random Access Memory (RAM), magnetic or optical disks.

5 **[00125]** Although the preferred embodiments of the present disclosure have been disclosed for illustrative purposes, those skilled in the art will appreciate that various modifications, additions and substitutions are possible, without departing from the scope and spirit of the disclosure as disclosed in the accompanying claims. Accordingly, the protection sought herein is as set forth in the claims below.

CLAIMS

1. A graphical development method comprising:

encapsulating, in each DAG component, an operator for implementing a function of the component, and a name of another DAG component allowed to be connected;
presenting a plurality of visualized DAG components to a user;
receiving selection of a plurality of DAG components by the user, and establishing a connectivity of the DAG components according to the function and the component allowed to be connected of the respective DAG component, thereby generating a DAG component graph.

2. The method of claim 1, wherein

the encapsulating further comprises: encapsulating one or more parameter or attribute configuration item; and

receiving selection of the plurality of DAG components by the user further comprises:

displaying a parameter or attribute configuration panel for the DAG component selected by the user;

receiving the configuration parameter or attribute data input by the user, and verifying the data in real time.

3. The method of claim 1, further comprising:

re-verifying, each time when the connectivity is changed by adding a DAG component into the connectivity, a downstream path of the added DAG component, wherein the downstream path includes the respective DAG components and the links formed thereby in the connectivity; or,

re-verifying, when any of the DAG components is clicked, links for the DAG components in a downstream path of the clicked DAG component.

4. The method of claim 3, wherein the re-verifying comprises: checking correctness of a field list and type of one or more input/output of each DAG component.

5. The method of claim 1, further comprising: selecting a region including a plurality of DAG components in the established connectivity, and saving the region as a callable template.

6. The method of claim 1, further comprising: analyzing code relationship of the DAG component graph after generating the DAG component graph.

7. The method of claim 6, wherein the analyzing comprises:

resolving the connectivity formed by the respective DAG components and links into a data structure;

decomposing the data structure into a plurality of logical operators according to a statement configuration;

generating statement codes correspondingly to the plurality of logical operators.

8. The method of claim 7, further comprising: receiving a switch instruction and performing an operation inverse to the analyzing to form the DAG component graph, after generating the statement codes.

5 9. A computer-readable storage medium storing a computer program which, when executed by a processor, implements the method according to any one of claims 1 to 8.

10. A computer device incorporated with the computer storage medium of claim 9.

11. A graphical development apparatus comprising:

an encapsulation module configured to encapsulate, in each DAG component, an operator for
10 implementing a function of a component, and a name of another DAG component allowed to be connected;

a presentation module configured to present a plurality of visualized DAG components to a user;

a graphics module configured to receive selection of a plurality of DAG components by the
15 user, establish a connectivity of the DAG components according to the function and the component allowed to be connected of the respective DAG component, thereby generate a DAG component graph.

12. The apparatus of claim 11, wherein the encapsulation module is further configured to encapsulate one or more parameter or attribute configuration item;

20 the graphics module is coupled to a parameter verification module, which is configured to display a parameter or attribute configuration panel for the DAG component selected by the user, and receive the configuration parameter or attribute data input by the user and verify the data in real time.

13. The apparatus of claim 12, wherein the graphics module is further coupled to a
25 re-verification module, which is configured to

re-verify, each time when the connectivity is changed by adding a DAG component into the connectivity, a downstream path of the added DAG component, wherein the downstream path includes the respective DAG components and the links formed thereby in the connectivity; or,

re-verify, when any of the DAG components is clicked, links for the DAG components in a
30 downstream path of the clicked DAG component.

14. The apparatus of claim 12, wherein the graphics module is further coupled to a template module, which is configured to select a region including a plurality of DAG components in the established connectivity, and save it as a callable template.

15. The apparatus of claim 12, wherein the graphics module is further coupled to an analysis

module, which comprises:

a first sub-module configured to resolve the connectivity formed by the respective DAG components and links into a data structure;

5 a second sub-module configured to decompose the data structure into a plurality of logical operators according to a statement configuration; and

a third sub-module configured to generate statement codes correspondingly to the plurality of logical operators.

10 16. The apparatus of claim 15, wherein the analysis module is coupled to a conversion module, which is configured to receive a switch instruction and perform an operation inverse to the analysis module to form the DAG component graph.