# IMPACT ANALYSIS

## BACKGROUND

This description relates to system analysis.

Computers are used to process large amounts of data. In general, the data is processed using computer programs what are written, at least in part, by computer programmers. These data processing systems can be complex.

Business and technical requirements can require that programs change. Implementing a change requires the allocation of personnel to make the change.

## SUMMARY

In general, aspect 1, a method includes the actions of receiving information about at least two logical data sets, the logical data set information identifying, for each logical data set, an identifier for at least one field in that logical data set, and format information about that field. The method includes the actions of receiving information about a transformation, the information identifying a first logical dataset from which the transformation is to receive data and a second logical data set to which the transformed data is to be provided. The method includes the actions of receiving one or more proposed changes to at least one of the fields of a logical data set. The method includes the actions of analyzing the one or more proposed changes based on information about the transformation and information about the first logical data set and the second logical data set. The method includes the actions of calculating one or more metrics of the proposed change based on the analysis. The method includes the actions of storing information about the one or more metrics.

Other embodiments of this aspect include corresponding computer systems, apparatus, and computer programs recorded on one or more computer storage devices, each configured to perform the action of the methods. A system of one or more computers can be configured to perform particular actions by virtue of having software, firmware, hardware, or a combination of them installed on the system that in operation causes the system to perform the actions. One or more computer programs can be

configured to perform particular actions by virtue of including instructions that, when executed by data processing apparatus, cause the apparatus to perform the actions.

The methods include an aspect 2 according to aspect 1 wherein the calculated metric provides a measure of direct impact. The methods include an aspect 3 according to aspects 1 or 2 wherein the calculated metric provides a measure an indirect impact. The methods include an aspect 4 according to aspects 1, 2, or 3 wherein the proposed change is one of the group consisting of a change in format of a field in a dataset or a change in a transformation. The methods include an aspect 5 according to aspects 1, 2, 3, or 4 wherein the transformation includes one or more rules to be applied to data from the first logical data set, and wherein analyzing the one or more proposed changes is further based on the one or more rules. The methods include an aspect 6 according to aspects 1, 2, 3, 4, or 5, wherein the method further includes the actions of Other embodiments of this aspect include associating a cost to the proposed change based on the metric.

Aspects can include one or more of the following advantages. The scope and cost of making a change can be estimated. Locations where a change will affect a program can be identified. Resources can be appropriately allocated.

Other features and advantages of the invention will become apparent from the following description, and from the claims.

## DESCRIPTION OF DRAWINGS

FIG. 1 illustrates an example of a data lineage for multiple executable programs.

FIG. 2 illustrate an example rule set and the inputs into the rules set.

FIG. 3 illustrates an example of a human readable rule set 300.

FIG. 4 illustrates a process by which a rule can be transformed from the human readable form into a transformation containing machine readable code.

FIG. 5 illustrates a report generator generating a report based on annotated code.

FIG. 6 shows an example of a data processing system in which the impact analysis techniques can be used.

FIG. 7 is a flowchart of an exemplary impact analysis procedure.

FIG. 8A is a block diagram of one embodiment of the invention showing the interrelationship of principal elements.

FIG. 8B is a block diagram of a data flow graph.

FIG. 9 is a block diagram of a typical graph having a rollup component and a sort component 904 with designated runtime parameters.

FIG. 10 is a diagram of one embodiment of a graphical dialog representing a runtime parameters grid that would be associated with a graph.

FIG. 11 is a flowchart that summarizes the process of using a runtime parameter.

FIG. 12 is a diagram of one embodiment of a graphical dialog generated by the key prompt.

FIG. 13 is a diagram of one embodiment of a graphical dialog generated by the filter prompt.

FIG. 14 is a diagram of one embodiment of a graphical dialog generated by the rollup prompt.

FIG. 15 is a diagram of one embodiment of a graphical dialog generated by the reformat prompt.

FIG. 16A is a block diagram of a first graph in which a MergeJoin component joins data from files A and B and outputs the result to an output file.

FIG. 16B is a block diagram of a second graph in which a Rollup component aggregates data from file A and outputs the result to an output file.

FIG. 16C is a block diagram of a graph in which a MergeJoin component joins data from files A and B, and a Rollup component aggregates the resulting data and outputs a final result to an output file.

FIG. 17 is a diagram of one embodiment of a graphical dialog presenting a Condition having a Condition-interpretation control.

FIG. 18 is a diagram of a graph showing a situation in which poisoning arises.

FIG. 19 is a flowchart that summarizes the process of runtime preparation of a graph that includes a Remove Completely conditional component.

FIG. 20 is a flowchart that summarizes the process of runtime preparation of a graph that includes a Replace With Flow conditional component for a particular embodiment of the invention.

FIG. 21 is a diagram of a graph representing a rollup application without runtime parameters.

FIG. 22 is a diagram of a graph representing a runtime parameterized version of the rollup application of FIG. 21.

FIG. 23 is a diagram of one embodiment of a graphical dialog representing a runtime parameters grid for the example application of FIG. 22.

FIG. 24A is a diagram of one embodiment of a graphical dialog representing a form generated by the Web Interface from the information in the parameters grid of FIG. 23.

FIG. 24B is a diagram of the form of FIG. 24A filled in by a user with parameter values.

FIG. 25 is a diagram of a graph representing a runtime parameterized rollup and join application.

FIG. 26 is a diagram of one embodiment of a graphical dialog representing a runtime parameters grid for the example application of FIG. 25.

FIG. 27 is a diagram of one embodiment of a graphical dialog representing a form generated by the Web Interface from the information in the parameters grid of FIG. 26.

FIG. 28 is a diagram of a graph representing a runtime parameterized rollup-join-sort application.

FIG. 29 is a diagram of one embodiment of a graphical dialog representing a runtime parameters grid for the example application shown in FIG. 28.

FIG. 30A is a diagram of a graph in which metadata is propagated.

FIG. 30B is a diagram of a sub-graph for a component in the graph of FIG. 30A.

FIG. 31 is a flowchart for a metadata propagation process.

FIG. 32A is a graph including parameters that have intra-component and inter-component dependencies.

FIGS. 32B and 32C are dependency graphs representing dependencies among the parameters of the graph in FIG. 32A.

FIG. 33 is a diagram of a modified topological sort process.

FIG. 34A is a graph illustrating the flow of data through a simplified sample example of a prior art payroll system.

FIG. 34B is a graph corresponding to the graph of FIG. 34A in accordance with the present invention.

FIG. 35 is a block diagram of a driver program in accordance with the present invention.

FIG. 36 is a flow diagram of a method for executing a graph in accordance with the present invention.

FIGS. 37A and 37B are flow diagrams of a step of the method illustrated in FIG. 36.

FIGS. 38A, 38B, 38C, 38D, 38E, 38F, 38G, and 38H are flow diagrams of steps of the method illustrated in FIG. 36.

FIG. 39 is a flow diagram of steps of the method illustrated in FIG. 36.

FIG. 40 is a flow diagram of steps of the method illustrated in FIG. 36.

FIG. 41 is a flow diagram illustrating the steps of inserting a source adapter in accordance with the present invention.

FIG. 42A is a block diagram of an illustrative graph.

FIG. 42B illustrates the insertion of a source adapter in accordance with the present invention.

FIG. 43 is a first phase of an example graph to which the invention is applied.

FIG. 44 is a second phase of an example graph to which the invention is applied

FIG. 45 is a second phase of an example graph to which the invention is applied.

FIG. 46 is a third phase of an example graph to which the invention is applied.

FIG. 47 is a fourth phase of an example graph to which the invention is applied

FIG. 48 is a fifth phase of an example graph to which the invention is applied.

FIG. 49 is a sixth phase of an example graph to which the invention is applied.

## DESCRIPTION

Generally, data processing systems can read data from a source, perform operations on the data to produce new data, and store the new data in a data store. Data processing systems range in complexity from trivial to extremely complex. In more complex systems, changes made to a data processing system can have wide ranging effects that can be difficult to determine. In order to determine the impact of making a

change to a system, it is helpful to determine the parts in the system that are directly affected by the change, and the parts in the system that are indirectly affected by the change. In general, parts in the system that are directly affected by the change may require an individual to manually adjust the system. For example, a programmer may be required to change the contents and behavior of an application. In general, parts of the system that are indirectly affected by the change may require that those parts be tested to ensure that the changes made by the programmer have not adversely affected the behavior of the application.

In order to determine, the scope of a change a system is analyzed to determine how data flows through the system. The system is also analyzed to determine how a change to either the data used by the system or the processing of that data could affect other parts of the system.

FIG. 1 illustrates an example of a data lineage 100 for multiple components. Components can include logical data sets and transforms. A transform can be, for example, dataflow graphs, java programs, compiled executable programs, or any combination thereof. In general, the transform can accept input data and can generate output data. For example, graph 1 104 accepts input data from logical data set 1 102 and logical data set 2 103 and generates output data that is provided to logical data set 2.

In general, logical data sets is representative of one or more physical data sets that store data. A physical data set may include unique data that can vary from day to day. In some implementations, a physical data sets with different data may be stored in separate files. For example, a dataset of foreign exchange rate data for November 8 may be stored in a file, "ExchangeRate_11_08", the dataset of foreign exchange rate data for November 9 may be stored in a file, "ExchangeRate_11_09", however, both the dataset for exchange rate data for November 8 and the dataset for exchange rate data for November 9 have common elements. For example, both datasets share a common data format. A logical data set enables a program or series of programs to be constructed to perform operations on the physical datasets independent of the particular contents of those datasets.

In general, information about logical data set includes information that attributes of the data set that does not vary with the data stored in the data set. For example, information about logical data sets may include field names, data types, record formats,

constraints, and other features. Logical data sets can be categorized as data sources and data sinks. A single logical data set can be a data sink for one transform and a data source for another transform. The corresponding physical datasets of a logical data set can be, for example a table in a relational database or a file on a file system (among other places). The data source can read the data records stored in the logical data set and can create in-memory data records. Components accept the in-memory data records created by the data source and modify or transform the data. Data values can be changed or transformed. New data records can be created. A data sink can provide an exit point from the dataflow graph and can store output records. Like the data source, a data sink can access, for example, a relational database table or a file stored on a file system. The components can be executed on a computer or other type of computer device. In other implementations, the execution of the dataflow graph can be distributed among multiple computing devices.

In some implementations, components can accept input data, for example on input ports, and produce output data, for example on output ports. Links connect an output port of a first component to an input port of a second component. Some components can have multiple input and output ports. The sequence of components and links by which a data record can navigate from an entry point to an exit point is referred to as a path. Data lineage can be used to identify the different paths and trace the flow of data through one or more components.

In this example the data element 'x'102 is a member of logical data set 1, the data element 'y' 103 is a member of logical data set 2. In general, a data element refers to individual records stored within a logical data set. A data element can be, for example, a logical data set can be a table in a relational database and a data element can be a row from that table. Data element 'x' and data element 'y' are input into graph 1 104. Graph 1 generate logical data set 2 106. Logical data set 2 contains data elements 'A' 108 and data element 'B' 110. These data elements are inputs into graph 2 112. Data element 'A' is used to generate data element 'C' 114. Data element 'C' is provided as an input into rule set 1 116. In general, a rule set is a collection of rules that are applied to data to generate an output. A rule set can be, for example, a series of tests and results that are applied to a value in a data element. A rule set can accept one or more inputs and based

on the values of those inputs produce one or more outputs. In general, a rule set can be compiled or made into a computer executable transformation. The data lineage graph 100 shown in FIG. 1 has been simplified for explanatory purposes and space considerations. In general, the presence of an ellipsis alone a line indicates that one or more components and data sources have been omitted. Data transformations may occur that are not shown. For example, data element 'A' 108 may be transformed to generate data element 'C' 114. Data element 'E' 118 may be transformed to generate data element 'G' 122, etc.

Rule set 1 generates two outputs data element 'E' 118 and data element 'F' 120. Data element 'E' 118 is used to generate data element 'G' 122. Data element 'G' is provided as input to rule set 2 130. Rule set 2 produces an output of data element 'I' 132. Data element 'I' is used to generate data element 'J' 140 of logical data set 3 138. Data element 'F' 120 is used to generate data element 'H' 124 and data element 'D' 126. Data element 'B' 110 is used to generate data element 'M' 128. Data element 'M' 128 and data element 'D' 126 are provided as input to rule set 3 134. Rule set 3 generates data element 'K' 136. Data element 'K' is used to generate data element 'L' 142 of logical data set 3 138. Data element 'Y' is provided as an input to rule set 4 144. Rule set 4 144 generates data element 'N' 146 of logical data set 3 138.

A change made to a logical data set or data element can affect many different rule sets and data elements. These changes can include, among other changes, a change to a schema or record format and changes to valid values for different data elements. For example, if a record schema of a data element is changed (for example, the record schema can change from a number to a string field) the change can affect each rule set that utilizes that data element and each rule set that utilizes a data element that depends on the changed data element. For example, a change made to the record format of data element C could affect rule set 1, data element E, data element F, data element G, data element H, data element D, rule set 2, data element I, data element J, rule set 3, data element K, and data element L. A change made to the record format of data element X could affect every other element in the data lineage (with the exception of data element Y, ruleset 4, or data element N).

A system can generate a report that provides information on the impact of a change to a data element or rule set. For example, the report 150 provides information on the impact of a change to data element A on graph 2.

The report 150 includes a direction, column. The direction column indicates a direction in the data lineage for which the report was generated. The direction can be either upstream, referring to rule sets, logical data sets, and data elements that precede the data element in the data lineage, or downstream, referring to rule sets, logical data sets, and data elements that follow the data element in the data lineage. For example, data element C is upstream of rule set 1 and downstream of data element A.

The report 150 also includes a graph column 154. The graph column identifies the graph that is the subject of a section of the report. In this example, graph 2 112 is the subject of the report. The report also includes a graph field column 156. The graph field column identifies the field that is the subject of the report. In general, the field will be an input to the graph if the direction is downstream and an output of the graph if the direction is upstream. In this example, the data element A 108 and B 110 is the subjects of the report.

The report 150 also includes a rule set field column 158. The rule set field column identifies data elements that are inputs (in the case of a downstream report) or outputs (in the case of an upstream report). The rule set column 160 identifies the rule set that is the subject of the row of the report. In this example, the report provides information about data element C as an input into rule set 1 (in the first tow 166), data element G as an input into rule set 2 (in the second row 168), data element H as an input into rule set 2 (in the third row 170), data element D as an input into rule set 3 (in the fourth row 172), and data element M as an input into rule set 4 (in the fifth row) 174.

The report 150 also includes a direct column 162 and an indirect column 164. The direct and indirect columns are determined by an analysis of the code, as described further below. However, they are presented here for completeness. The direct column reports the number of times that the data element identified by the rule set field is directly referenced within the rule set. For example, the direct column can include a count of the expressions that directly assign a value to the output. The indirect column 164 identifies the number of times that the data element identified by the rule set field affects the value

of one or more other data elements within the rule set identified by the rule set field. For example, the indirect column can display a count of the total number of rule cases or other expressions that contribute to the output value of the data element. For an output that is computed by a business rule, the expression count is the number of rule cases, including a default value if there is one. In this example, data element 'C' is directly references 13 times in rule set 1 and affects the value of one or more other data elements 70 times.

In order to generate the report 150, a system processes a rule set to determine which data elements are relevant to the rule set. The rule set can be expressed as a set of criteria that can be used, for example, for converting data from one format to another, making determinations about data, or generating new data based on a set of input data.

FIG. 2 illustrate an example rule set and the inputs into the rules set. As described above, rule set 3 134 has two inputs, data element 'D' 126 and data element 'M' 128. Rule set 3 134 may reference the inputs as a "Years" parameter 202 and an "Income" parameter 204. In this example, the "Years" parameter 202 and the "Income" parameter 204 are processed by a transform 206, described in more detail below. The Transform 206 produces a "Risk" output 208 which is provided as data element 'D' 136.

FIG. 3 illustrates an example of a human readable rule set 300. The human-readable rule set 300 can be defined using a graphical user interface (GUI) or can be defined in a flat file or other structure. The human-readable rule set 300 can be later compiled into a transform, for example, the transform 206 of FIG. 2, as described below. For example, the human-readable rule set 300 can be compiled into the rule set 3 134 of FIG. 1. For example, if the data element D represented income and the data element M represented years as a customer. Referring again to FIG. 3, the human-readable rule set 300 is illustrated in a tabular form. The human-readable rule set 300 presented in FIG. 3, can be used to determine a risk category 310 as an output 304 based on two inputs 302: years 306 and income 308. In this example, there are seven potential conditions. A first rule 312 states if a person has been a customer for more than 15 years, the risk is low, regardless of income. A second rule 314 states that if the customer has an income in excess of $150,000 the first is low, regardless of the number of years the person has been a customer. A third rule 316 states that if the person has been a customer for more than

ten years (but less than 15) and has income of more than $60,000, the risk is low. A fourth rule 318 states that if the person has been a customer for more than 5 years, the risk is medium regardless of income. A fifth rule 320 states that if the person has an income of greater than $50,000 then the risk is medium regardless of the amount of time the person has been a customer. A sixth rule 322 states that if the person has been a customer for more than three years and has an income of more than $40,000 then the risk is low. A seventh rule 324 is a catch all rule that states that otherwise the risk is high.

It is noted that, in this example, the rules are evaluated sequentially. Once a person qualifies for a risk category then rule processing is complete. For example, if a person has been a customer for more than 15 years, and is assigned a risk of "low" (from row 312) then the remaining rows will never execute.

In order to determine the impact of a change to one of the input or output fields, a system can perform an analysis of the rule set as described below.

To implement rule set in a graph-based computation environment, a transform is generated which receives input records from one or more data sources, e.g., data element 'C' 106, and inserts a data element into an output dataset, e.g. data element 'E' 118 and data element 'F' 120. Input and output datasets can also be referred to as data streams. As shown in FIG. 1, the transforms can then be implemented in graph-based computations having data processing components connected by linking elements representing data flows.

FIG. 4 illustrates a process by which a human readable rule set can be transformed into a transform containing machine readable code. A rule set 402, for example the human-readable rule set 300 of FIG. 3, is provided to a rule generator 408. The rule generator compiles the rule set 402 into an intermediate form. For example, the rule generator 406 can generate annotated code 408. The annotated code 408 can include reported metrics 410 which defines the direct and indirect impact of a change to a rule. For example, the human-readable rule set 300 can result in the generation of annotated code, such as

```
/* Default Risk Rules */
/*@
Reported Metrics:
[
Output references:
[default risk, 7, 0]
```

```
Input references:
[income, 1, 6]
[years, 1, 6]
]
@*/

if (years > 15) {
      default_risk = "Low"
} else if (income > 150000) {
      default_risk = "Low"
} else if (years > 10 && income > 60000) {
      default_risk = "Low"
} else if (years > 5) {
      default_risk = "Medium"
} else if (income > 50000) {
      default_risk = "Medium"
} else if (years > 3 && income > 40000) {
      default_risk = "Medium"
} else {
      default_risk = "High"
}
```

As discussed above, a direct impact describes the number of times that the data element identified by the rule set field is directly referenced or set within the rule set. The indirect impact identifies the number of times that the data element identified by the rule set field affects the value of one or more other data elements within the rule set.

A rule generator 406 can generate the direct and indirect metrics can occur in a variety of ways. For example, in some implementations, the rule generator 406 can analyzing the rule set to identify each time a data element is accessed and each time another value depends on the data element. More complex scenarios can also be tracked. The rule generator 406 can track every variable that depends on the value of the input or output value regardless of how indirect. For example, if a variable affects an intermediate value and the intermediate value affects a final value, the system can record both the intermediate value and the final value as indirect effects. For example, the human readable rule set 300 has four rules that access the value of the years 306 input and four rules that access the value of the income 308 input, and seven rules that set the value of the risk 310 output. In some implementations, a rule set may be presumed to set the value for each parameter at least once. For example, the years input is set when the input value is provided to the rule set.

In some implementations, the rule generator 406 can count the number of rules in the rule set that depend, at least in part, on a parameter. For example, the human readable

rule set 300 includes seven rules that depend on the years 306 input, seven rules that depend on the income 308 input, and seven rules that set the risk 310 output. As discussed above, rule 324 it a catch all rule. In some implementations, catchall rules may be ignored by the rule generator 406.

The annotated code 408 can be provided to a rule compiler 412. The rule compiler 412 can compile the annotated code 408 into the transform 206. In general, a transform is a machine (or virtual machine) executable program, for example, the executable program 416.

Referring to FIG 4, a report generator can generate a report 408 based on annotated code 402 for a data lineage 4. For example, referring back to Fig. 1, the system can use the data lineage to determine that, for example, data element X affects the value of data elements A, B, C, D, E, F, G, H, I, J, K, L, and M. Therefore when processing graph 2, the system determines that a change to data element X implicates rule set 1, rule set 2, and rule set 3. However, data element X does not implicate rule set 4. Therefore, the report generator can determine that rule set 4 does not need to be analyzed as part of the impact analysis of a change to data element X.

FIG. 5 illustrates a report generator generating a report based on annotated code. The report generator 506 identifies which inputs would be affected by a change and records the results from the computed metrics of impact. For example, data element 'X' affects both inputs to rule set 3. Therefore, the report generator records the metrics (410 of FIG. 4) in the report (such as rows 172, 174 of FIG. 1).

In some implementation, costs can be associated with each of the direct and indirect tallies. For example, it can be determined that a direct effect would require a predetermined amount of a programmers time and a predetermined amount of a quality assurance persons time. Similarly, it can be determined that an indirect effect would require a predetermined amount of a quality assurance persons time. Based on the predetermined times, a tally of the direct and indirect effect, and a cost associated with a computer programmers time and a quality assurance persons time, the system can provide an estimate of the cost to make the change to the analyzed system.

In some implementations, the rule generator can be used to assist a developer in identifying different portions of a system, for example, a system represented by the data lineage 100 of FIG. 1.

FIG. 6 shows an example of a data processing system 600 in which the impact analysis techniques can be used. The system 600 includes a data source 602 that can include one or more sources of data such as storage devices or connections to online data streams, each of which can store or provide data in any of a variety of formats (e.g., database tables, spreadsheet files, flat text files, or a native format used by a mainframe). An execution environment 604 includes a rule generator 606 and a report generator 612. The execution environment 604 can be hosted, for example, on one or more general-purpose computers under the control of a suitable operating system, such as a version of the UNIX operating system. For example, the execution environment 604 can include a multiple-node parallel computing environment including a configuration of computer systems using multiple central processing units (CPUs) or processor cores, either local (e.g., multiprocessor systems such as symmetric multi-processing (SMP) computers), or locally distributed (e.g., multiple processors coupled as clusters or massively parallel processing (MPP) systems, or remote, or remotely distributed (e.g., multiple processors coupled via a local area network (LAN) and/or wide-area network (WAN)), or any combination thereof.

The rule generator module 606 reads rule specifications from the data source 602 and stores annotated code for the rules. Storage devices providing the data source 602 can be local to the execution environment 604, for example, being stored on a storage medium (e.g., hard drive 608) connected to a computer hosting the execution environment 604, or can be remote to the execution environment 604, for example, being hosted on a remote system (e.g., mainframe 610) in communication with a computer hosting the execution environment 604, over a remote connection (e.g., provided by a cloud computing infrastructure).

The report generator 612 uses the annotated code generated by the rule generator 606 and data lineage, which can be stored in the data source 602, to generate a report of the impact of making a change. The output data can be 614 stored back in the data source 602 or in a data storage system 616 accessible to the execution environment 604,

or otherwise used. The data storage system 616 is also accessible to a development environment 618 in which a developer 620 is able to determine the effect of making a change to a data element, rule, of other programming construct. The development environment 618 is, in some implementations, a system for developing applications as dataflow graphs that include vertices (representing data processing components or datasets) connected by directed links (representing flows of work elements, i.e., data) between the vertices. For example, such an environment is described in more detail in regards to Managing Parameters for Graph-Based Applications. A system for executing such graph-based computations is described below. Dataflow graphs made in accordance with this system provide methods for getting information into and out of individual processes represented by graph components, for moving information between the processes, and for defining a running order for the processes. This system includes algorithms that choose interprocess communication methods from any available methods (for example, communication paths according to the links of the graph can use TCP/IP or UNIX domain sockets, or use shared memory to pass data between the processes).

FIG. 7 is a flowchart of an exemplary impact analysis procedure 700. The process can be performed by a data processing system, such as the data processing system 600 of FIG. 6.

Information about two data sets is received (702). The logical data set information can identify, for each logical data set, an identifier for at least one field in that logical data set, and format information about that field.

Information about a transformation is received (704). The information can identify, from the two logical data sets, a first logical dataset from which the transformation is to receive data, and a second logical dataset to which the transformed data is to be provided. The transformation may include information about one or more rules to be applied to data from the first logical data set, and wherein analyzing the potential impact of the one or more proposed changes is further based on the one or more rules.

One or more proposed changes is received (706). A proposed change can be a change to a format of a field in a dataset, a change to a transformation, or a change to a rule set. In some implementations, the proposed change identifies the field in a logical

data set or the transformation that is to be altered without specifying the nature of the change. For example, the proposed change can specify that field 'X' is to be changed without indicating that the change is from a decimal record format to a string record format.

The proposed change is analyzed (708).

A metric of proposed change is calculated (710). The metric can measure the impact of the change. The metric can include a measure of direct impact and / or a measure of indirect impact. Examples of measures of direct impact include, but are not limited to, locations in a rule set where a changed input parameter is accessed. Examples of measure of indirect impact include, but are not limited to, locations in a rule set where values are set based on a value in a changed input parameter.

The metrics are stored (712). The metrics can be stored in a flat file, a relational database, or in any other persistent data store. The metrics may be stored in the form of a report. The report can be generated identifying the metric of impact. The report can associate the measures of direct impact and the measures of indirect impact with particular portions of the data lineage. For example, the report can indicate that a particular data flow graph, data flow graph field, rule set field, or rule set is associated with a measure of direct impact and a measure of indirect impact.

In some implementations, the report can be tied into the data lineage, for example, through Hyper-Text Transport Protocol (HTTP) links. Selecting or clicking on the link can navigate a browser on a client device to an application or website that allows the user to view a particular portion of the data lineage. For example, referring to FIG. 1, selecting or clicking on the third row 170 can cause a browser or other application on the client device to display the dataflow graph 'Graph 2' 112. In some implementations, that particular graph, graph field, rule set field, and rule set can be visually distinguished, for example, by highlighting.

In some implementations, the report can include an average development and testing cost that can be associated with the proposed change. For example, the report can associate a dollar cost with a direct change and a dollar cost with an indirect change. In some implementations, the dollar cost can be a parameter provided to the process. In other implementations, a default value can be associated with each change. For example,

a direct change can be determined to cost $100 and an indirect change can be determined to cost $25.

Managing Parameters for Graph-Based Applications

FIG. 8A is a block diagram of one embodiment of the invention showing the interrelationship of principal elements. A graphic development environment (GDE) 802 provides a user interface for creating executable graphs and defining parameters for the graph components. The GDE may be, for example, the CO>OPERATING SYSTEM® GDE available from the assignee of the present invention. The GDE 802 communicates with a repository 804 and a parallel operating system 806. Also coupled to the repository 804 and the parallel operating system 806 are a Web Interface 808 and an executive 810.

The repository 804 preferably is a scalable object-oriented database system designed to support the development and execution of graph-based applications and the interchange of metadata between the graph-based applications and other systems (*e.g.*, other operating systems). The repository 804 is a storage system for all kinds of metadata, including (but not limited to) documentation, record formats, transform functions, graphs, jobs, and monitoring information. Repositories are known in the art; see, for example, U.S. Patent Nos. 5,930,794; 6,032,158; 6,038,558; and 6,044,374.

The parallel operating system 806 accepts the representation of a data flow graph generated in the GDE 802 and generates computer instructions that correspond to the processing logic and resources defined by the graph. The parallel operating system 806 then typically executes those instructions on a plurality of processors (which need not be homogeneous). A suitable parallel operating system is the CO>OPERATING SYSTEM® available from the assignee of the present invention.

The Web Interface 808 provides a web-browser-based view of the contents of the repository 104. Using the Web Interface 808, a user may browse objects, create new objects, alter existing objects, specify application parameters, schedule jobs, etc. The Web Interface 808 automatically creates a forms-based user interface for a parameterized graph based on information stored in the repository 804 for the graph's runtime parameters.

The executive 810 is an optional repository-based job scheduling system accessed through the Web Interface 808. The executive 810 maintains jobs and job queues as

objects within the repository 804, and the Web Interface 808 provides a view of and facilities to manipulate jobs and job queues.

FIG. 8B shows a simple data flow graph 820 with an input dataset 822 connected by a flow 824 to a filter component 826. The filter component 826 is connected by a flow 828 to an output dataset 830. A dataset can include, for example, a file or a database table that provides data (e.g., an input dataset) or receives data (e.g., an output dataset) for a computation performed by a data flow graph.

The flow of data represented by a "flow" in a data flow graph can be organized into discrete data elements. For example, the elements can include records from a dataset that is organized into records (or rows) and fields (or columns). Metadata describing the sequence of fields and data types corresponding to values in a record is called a "record format."

Components and datasets in a graph have input and/or output ports for connecting to flows. The "source ends" of the flows 824 and 828 interface with an output port of the input dataset 822 and with an output port of the filter component 826, respectively. The "sink ends" of the flows 824 and 828 interface with an input port of the filter component 826 and with an input port of the output dataset 830, respectively. An input or output port of a dataset or component is associated with metadata, such as a record format for the data flowing into or out of the port.

A parameter including a record format for a port or other metadata associated with a component is bound to a value according to rules for parameter scoping. A parameter can be bound to a value at design time or at runtime (i.e., a "runtime parameter," as described below). The value of a parameter can be defined, for example, by a user over a user interface (e.g., in response to a prompt), defined from a file, or defined in terms of another parameter in the same context or a in different context. For example, a parameter can be exported from a different context (e.g., a parameter evaluated in the context of a different component) by designating the parameter to have a "same as" relationship to another parameter.

A component used in a graph can be implemented using other components that are interconnected with flows forming a "sub-graph." Before a sub-graph is used as a component in another graph, various characteristics of the component are defined such as

the input and/or output ports of the component. In some cases, characteristics of a component having to do with relationships among sub-graph components should be specified before the component is used in a graph. For example, a prompting order for runtime parameters of sub-graph components may need to be selected. An approach for selecting a prompting order for runtime parameters of components in a graph is described in more detail below.

*Metadata Propagation*

The value of metadata associated with a port, such as a record format parameter, can be obtained by "propagation." Metadata propagation can occur "externally" or "internally." For external metadata propagation, the value of a record format parameter for a port of a first component can obtain a value by propagating a record format value for a port of a second component that is connected to the first component by a flow. The value is able to propagate either downstream from the source end to the sink end of a flow or upstream from the sink end to the source end of a flow. Metadata propagates from a port that has defined metadata to a port that does not have defined metadata.

For internal metadata propagation, metadata defined for one port of a component propagates to another port of that component based on a sub-graph that implements the component. In some cases, internal metadata propagation occurs over "non-transforming" internal data paths. For example, a user may provide metadata for the input port of a sort component that specifies the data type of records flowing into the sort component. Since the sort component re-orders but does not transform the records, the data type is not changed by the sort component and the data type propagates unchanged to the output port of the sort component accurately describing the data type of the records flowing out of the sort component.

Some components do transform (or optionally transform) data flowing through them. For example, a user may provide metadata for the input port of a filter component that specifies the fields of records flowing into the filter component. The filter component may remove values of a given field from each record. A metadata definition can be used to specify that the metadata for the output port of the filter component is related to the metadata of the input port according to the filter action of the component. For example,

the filtered field may be removed from the metadata specifying the record fields. Such a metadata definition can be supplied even before the input port metadata is known. Therefore, metadata can propagate even over transforming internal data paths by allowing metadata associated with a port to be specified as a function of one or more parameters, including metadata for another port, as described in more detail below.

This internal and external metadata propagation can optionally be configured to occur at design time while a graph is being constructed and a user supplies metadata for some ports of some components in the graph. Alternatively, metadata propagation can occur after a graph is constructed, including at or just before runtime.

*Runtime Parameters*

A runtime parameter allows an application builder to defer the value of a parameter setting (e.g., the key parameter of a sort function, file names, record formats, transform functions, etc.) to runtime (e.g., the time a program is executed or soon to be executed on a computer system). The values of runtime parameters may be supplied by the end user or be derived from a combination of other runtime parameters or objects stored in an object repository.

Runtime parameters add a certain amount of flexibility to an application. Additional flexibility is achieved by using those parameters to compute metadata (data formats or types, and program logic or transforms) on demand. Types and transforms may be synthesized from other types and transforms, user-supplied parameter values, and stored objects (e.g., from a repository). This makes it possible to build "generic" applications that work on input data of any type, or that produce data through a series of transforms whose construction is controlled, directly or indirectly, through runtime parameter values.

In some implementations, when creating or editing a runtime parameter, a developer may specify a prompt for each parameter and the conditions for displaying the prompt. The system interprets the prompting directives to present, if conditions are met, a graphical user interface (GUI) control for receiving the parameter value.

*Designation of Runtime Parameters*

Runtime parameters provide a mechanism for a developer to modify the behavior of a graph based on external input at graph execution time (i.e., runtime). In the preferred embodiment, these external values are provided by direct user input. However, these external values also may come from a number of different sources, including environment variables and command line parameters. The GDE 802 generates the correct code to handle all of these situations as well as prompting the developer for test values when the graph is executed directly from the GDE. Using runtime parameters, a developer can, for example, explicitly declare that the path of an input file will be provided by an environment variable with a particular name; that environment variable then becomes a known part of the graph's interface. Thus, there is a well-defined interface to such parameters. There is no need, for example, to read a generated shell script and search it for references to environment variables and command-line arguments to find the set of parameters that control the execution of a particular graph.

FIG. 9 is a block diagram of a typical graph 900 having a rollup component 902 and a sort component 904 with designated runtime parameters. The runtime parameters (a key for the sort component 904 and rules for the rollup component 902) would be presented to a user in an interface 906 for input. The following sections describe how to designate a runtime parameter, and create an integrated user interface for presentation of runtime parameters prompting for user input.

A runtime parameter may be designated or defined in a number of ways. One way is by use of a runtime parameters grid displayed in the GDE 802. FIG. 10 is a diagram of one embodiment of a graphical dialog representing a runtime parameters grid 1000 that would be associated with a graph. A new runtime parameter is created by simply filling in the appropriate fields. An object associated with each runtime parameter is created in the repository 804 and linked to all graph components that utilize the parameter. For example, if a sort key for a graph sort component is defined as a runtime parameter, an object representing the sort key parameter is stored in the repository 804 and linked to the associated sort component. An alternative way of defining a runtime parameter is to specially flag an existing parameter of a graph component and make it "visible" (export it) to other components. A combination of these methods may be used. For example, when creating a component, a developer may designate a particular parameter of that

component as a runtime parameter. The developer may then use a parameter grid to set default values and other characteristics of all of the runtime parameters for a graph, and define new runtime parameters.

When the graph is run, the parameters are processed to obtain values for each parameter from user input or from external programmatic sources (e.g., command line parameters or environmental variables). In the illustrated embodiment, the runtime parameters grid 1000 includes the following fields:

*Name* 1002 – This field contains the name of the runtime parameter. "Score_threshold" is the example shown for a name.

*Type* 1004 – This field contains the type of value to be allowed in the runtime parameter. "Integer" is the example shown for a type. Supported types in the illustrated embodiment are:

- boolean – value can be either True or False;
- choice – value is one of a list of values;
- collator – a key parameter value;
- dataset – an external data file name and location;
- date – a date value;
- expression – an arithmetic, logical, and/or conditional expression (e.g., a select expression);
- float – a floating point number;
- integer – an integer number;
- layout – a parallel or serial layout definition;
- record format – a record description or a file containing a record description;
- string – an arbitrary character string;
- transform – a transform description or a file containing a transform description.

Location (Loc) 1006 – This field is used with record format and transform types. It specifies whether the type field 1004 describes a file location or whether it contains an embedded description. Supported locations are:

- Embedded – the parameter will contain the record or transform description;
- Host – the parameter will contain a reference to a file on a host machine;
- Local – the parameter will contain a reference to a file on a local machine;

- Repository – the parameter will contain a reference a repository transform or record format.

*Default Value* 1008 – This field contains either (1) the default value for the runtime parameter which will be used if no other value is provided from an external programmatic source, or (2) a rule or expression describing how to derive the runtime value from user input or how to obtain that information interactively from the user executing the graph. In the latter case, a second default value field (not shown) may be used to provide a value for the runtime parameter if the user does not provide an input value. For types of "boolean" and "choice", this field limits the user to the valid choices. For "layout" types, this field is read-only and displays the currently defined layout definition. For all other types, this field preferably is a simple text editor into which the user may type a valid string.

*Edit* 1010 – Clicking on the edit space 1010 (or an icon; for example, a pencil icon) in a parameter row will bring up a more advanced edit window, which walks a user through the various options for editing the default value field 1008. In the illustrated embodiment, the following editors are available for their associated types:

- Single line edit – for integer, float, date and string types;
- Choice dialog – for boolean and choice types;
- Key Editor – for a collator type;
- File Browser – for a dataset type and for record format and transform types where the location is not embedded;
- Transform Editor – for a transform type with a location of Embedded;
- Record Format Editor – for a record format type with a location of Embedded;
- Expression Editor – for an expression type;
- Layout Editor – for a layout type.

The above editors are launched unless the Kind field value (see below) is "PL" (for Parameter Language). In this case the user is presented with an editor with which to define the rules for deriving or prompting for the parameter value at graph execution time.

*Description* 1012 – This is a free format field in which a developer describes the expected values of the runtime parameter. It is used as a prompt at runtime if the default value contains a rule for asking the user for an input value.

*Kind* 1014 – This field defines where a graph is to obtain the value for the associated parameter at graph execution time. Supported kind field 1014 values are:

- Environment – The value for the runtime parameter is expected to be found in an environment variable of the same name. If the environment variable is not defined, then the value in the default value field 1008 is used. If the parameter is required (i.e., an exported parameter), and the default value field 1008 is empty, then a runtime error will be generated and graph execution will stop.
- Positional – The value for the runtime parameter is expected at its relative position on a command line invoking the application. For example, if a runtime parameter is the third positional runtime parameter defined, then its parameter value will be expected as the third positional command line argument in an execution script. Any specified positional parameters must be provided and a runtime error will be generated if one is missing.
- Keyword – The value for the runtime parameter is expected as a keyword command line parameter. In the illustrated embodiment, keyword parameters are of the form:

  -<parameter name> <parameter value>.

  Keyword parameters are optional and a runtime error will only be generated if the keyword parameter is not provided and the default value field 1008 is blank and a corresponding exported parameter is required.
- Fixed – The runtime value for the parameter is always the default value. This is useful for sharing a constant value between two or more runtime parameters.
- PL – The default value of the runtime parameter contains a PL expression which will be interpreted at graph execution to either derive the value of the runtime parameter from other parameters or prompt the user for additional input. The Component Description Language that is selected for use with any particular embodiment of the invention may be any suitable scripting language, such as the publicly available object-oriented scripting language "Python". Such scripts can construct metadata (types and transforms) under

program control, and perform conditional tests, comparisons, data transformations, arithmetic and logical operations, string and list manipulations, and other functions on user input, externally programmatically supplied input, and other runtime parameters to generate a final value for any runtime parameter.

In the illustrated embodiment, a useful convention for referencing a runtime parameter that has been created directly on the runtime parameters grid 1000 is to simply enter the parameter name preceded by the dollar sign "$". For example, $key references a runtime variable named key. In the illustrated embodiment, new runtime parameters default to a type of "string" and a default kind based on the value in the advanced options dialog for the default runtime kind (the default runtime kind is "Environment").

Because runtime parameter values can are determined at runtime, and PL scripts can provide conditional testing, "conditional" runtime parameters can be created. A conditional runtime parameter causes a prompt to be generated for user input only if all of the conditions for the parameter – determined at runtime – are enabling. Thus, for example, if a user responds to a first prompt requesting whether a data set is to be sorted with "NO", a second, conditional prompt that requests a sort key need not be displayed.

Thus, during a design phase ("design time"), a developer designates a particular parameter of a graph component as a "runtime" parameter. An object associated with that graph component is then stored with the relevant parameter data (e.g., the types of information from the parameters grid 1000 of FIG. 9).

FIG. 11 is a flowchart that summarizes the process of using a runtime parameter. During runtime, parameter objects corresponding to an application to be executed are retrieved (e.g., from a repository) (STEP 1100). A determination is made for each such object as to whether user input is indicated (STEP 1102). If so, a determination is made as to whether any condition for display of the prompt has been met (STEP 1103), which may include evaluation of user input to prior prompts. If not, a default value is used (STEP 1108). Alternatively, the parameter value may not be needed (e.g., a sort key would not be needed if the user did not choose to activate a sort function), and thus may be ignored. Otherwise, a prompt is generated for user input (STEP 1104).

If the user does not input a value for a particular parameter (STEP 1106), the default value for the parameter may be selected (STEP 1108). Alternatively, an error

condition may be raised to indicate the lack of user input. In any event (assuming no error condition because of a lack of user input), a determination is made of the final value for the parameter, taking into account transformations of the input and dependencies and conditions based on other parameters (STEP 1110).

If a determination is made that user input is not indicated for a particular parameter (STEP 1102), a determination is then made as to whether the parameter value is to be externally supplied programmatically, such as by an environment variable or a command line parameter (STEP 1112). If not, the default value for the parameter is selected (STEP 1114). Alternatively, an error condition may be raised to indicate the lack of available input of the specified type. In any event (assuming no error condition because of a lack of external input), a determination is made of the final value for the parameter, taking into account transformations of the input and dependencies and conditions based on other parameters (STEP 1110).

Once the final parameter values are determined, as an optional step all conditional components (discussed below) can be removed either completely or replaced by flows (i.e., a graph link or edge), according to the specified conditions and the rules outlined above (STEP 1116). Once the operational graph structure is finalized and the final parameter values are determined, the graph is executed in conventional fashion (STEP 1118).

*Test Values*

In order to support a developer during the creation and testing of graphs with runtime parameters, the preferred embodiment of the GDE 802 also supports test values for runtime parameters. When a developer runs a graph with runtime parameters or wants to view the underlying code affecting a graph component, the GDE 802 displays an associated test parameters grid where the user can enter new test values for one or more runtime parameters. Preferably, the last set of test values used is remembered and saved with the graph.

For each runtime parameter, the developer enters a desired test value in a test value column. An edit field may be associated with each test value column. The test

value field and edit field behave the same as the default value field and edit field in the runtime parameters grid 900 except when the parameter kind is PL.

If a PL expression indicates that the user is to be prompted for a value for a particular runtime parameter, then the test value field and the edit behavior are based on the interpretation of the associated PL expression. If the PL expression simply derives a value based on other input, then in normal mode the runtime parameter is not visible in the test values grid.

*Specifying How Runtime Parameters Get Their Values*

After a parameter has been designated as a runtime parameter, a corresponding object is created in the repository 804. If the runtime parameter has a kind field 914 value of "PL", the default value field 1008 for the parameter includes a `prompt_for` pseudo-function with the following preferred form:

```
prompt_for "prompt-kind[modifiers]" options ...
```

As indicated above, the `prompt_for` pseudo-function may be part of a conditional expression that determines whether a prompt is to be displayed based on prior input.

For such objects, a user interface is used to present direct entry runtime parameters to a user. In the preferred embodiment, the Web Interface 808 provides this function. In particular, during runtime, each `prompt_for` pseudo-function of each - runtime parameter object is parsed by the Web Interface 808 to generate a web page (e.g., in HTML) having a corresponding user prompt. (Alternatively, such web pages can be generated before runtime and simply presented at runtime. However, runtime generation of such web pages provides greater flexibility. In particular, the contents of a page can depend on prior user input.) The Web Interface 808 is used in conjunction with a conventional web browser that can display such web pages and receive user input.

The `prompt_for` pseudo-function indicates to the Web Interface 808 how to prompt for a parameter value. In particular, the `prompt-kind` parameter, a string constant, indicates what kind of user interface (UI) element to present (text box, dropdown list, etc.). The *modifiers* part of the string, a comma-separated list of keywords,

provides some options common for various kinds of prompts. In the illustrated embodiment, space is not significant within the modifiers string. Modifier keywords are interpreted as follows:

- The keyword `in place` declares that the element should be presented directly at the summary level user interface for an application, allowing the value to be supplied without "drilling in" to a lower level. If `in place` is not specified, a simple "edit" button is presented at the summary level interface which will takes a user to another page to supply the parameter value.

- The keyword `blank ok` declares that a user need not supply a value; the application will deal with the default value in a reasonable way. If `blank ok` is not specified, then the user will not be able to execute the application without supplying some value.

Following are some examples of prompt_for calls with different kinds of modifiers:

```
${prompt_for "text,inplace"}
${prompt_for "filter, in place", $input_type}
${prompt_for "radio, blankok, in place", ${list 1, 2, 3}}
```

The remainder of this section lists a variety of `prompt`-kinds and their corresponding `options` and explains how each would appear in a web page generated by the Web Interface 808.

`text` *[size]* – Presents a conventional single-line text box `size` characters wide (if *size* is not supplied it defaults to the browser's default size for text boxes).

radio choice-list [description-list] – Presents a conventional "choose one" prompt in the form of a set of radio buttons, one button for each element of the *choice-list*. If *description-list* is supplied, each choice is labeled with the corresponding description; otherwise, the choices are labeled with the string form of the corresponding item from the *choice-list*.

`radioplus` *choice-list [description-list]* – Like radio, but presents an additional button next to a text box, to allow a user to choose a "write-in" value not in the choice-list.

`checkbox` *choice-list [description-list]* – Presents a conventional "choose zero or more" prompt in the form of a set of check boxes, one button for each element of the choice-list. If description-list is supplied, each choice is labeled with the corresponding description; otherwise, the choices are labeled with the string form of the corresponding item from the choice-list.

`dropdown` *choice-list [description-list, size]* – Presents a conventional "choose one" prompt in the form of a dropdown list for the elements of the choice-list. If description-list is supplied, each choice is labeled with the corresponding description; otherwise, the choices are labeled with the string form of the corresponding item from the choice-list. If size is supplied, that many choices will be visible at once; otherwise, only one will be visible.

`multidropdown` *choice-list [description-list, size]* – Presents a conventional "choose zero or more" prompt in the form of a dropdown list for the elements of the choice-list. If description-list is supplied, each choice is labeled with the corresponding description; otherwise, the choices are labeled with the string form of the corresponding item from the choice-list. If size is supplied, that many choices will be visible at once; otherwise, the browser's default number of items is shown.

`key` *type-obj [size]* – Presents a prompt for a key (also known as a collator) made up of fields from the given type-obj. The key can have as many as size parts, which defaults to the number of fields in type-obj. FIG. 12 is a diagram of one embodiment of a graphical dialog 1200 generated by the key prompt. Following is an example of the script text for a 3-entry key prompt, where the file */datasets/fixed* defines the contents of the available keys shown in the drop down boxes 1202:

${prompt_for "key", ${dataset_type "/datasets/fixed"},3}

In the illustrated embodiment, the normal collation order is ascending, but a user can select a descending collation order for a key by checking an associated check box 1204.

`filter` *type-obj* – Presents a prompt for a filter expression made up of conditions on each field of the given *type-obj*. The `blank ok` modifier has no effect for filters; a blank filter yields a "True" expression. FIG. 13 is a diagram of one embodiment of a graphical dialog 1300 generated by the filter prompt. The available field names 1302

associated with each expression text edit box 1304 are defined by *type-obj*. Comparison values are entered into the text edit boxes 1304, and a comparison operator (e.g., equal, greater than, less than or equal to) is selected from a corresponding dropdown list control 1306.

`flexifilter` *type-obj* – Similar to the filter prompt, but presents a prompt for a filter expression made up of conditions on each field of the given *type-obj* where the field name on each line is selectable from a dropdown list. This permits using the same field for multiple conditions (e.g., field STATE = MA OR field STATE = CA).

`rollup` *type-obj key [size]* – Presents a prompt for a rollup computation based on the fields of the given *type-obj* being rolled up by the given key. The rollup can have as many as size rules, which defaults to the number of fields in type-obj. The blank ok modifier has no effect for rollups; a blank rollup yields a package that provides just the key value for each group. FIG. 14 is a diagram of one embodiment of a graphical dialog 1400 generated by the rollup prompt. In the illustrated embodiment, a column of dropdown boxes 1402 defines the available rollup computation functions (e.g., sum, minimum, maximum). The available field names 1404 associated with each computation are defined by type-obj. Each rollup rule has an associated text edit box 1406 for user definition of a desired expression, a "where" text edit box 1408 for defining (through a boolean expression) criteria under which the source value will take part in the computation, and an output field text edit box 1410 for designating a field that will receive the computation result. In cases where it can be unambiguously derived, the name of the output field need not be specified.

`reformat` *type-obj* [size] – Presents a prompt for a reformat computation based on the fields of the given type-obj. The reformat can have as many as size rules, which defaults to the number of fields in type-obj. FIG. 15 is a diagram of one embodiment of a graphical dialog 1500 generated by the reformat prompt. In the illustrated embodiment, the `reformat` prompt includes a section 1502 for simply copying input fields to like-named output fields (either selected/deselected individually using checkbox controls or collectively by using Select All or Select None buttons). A second section of the prompt includes a column of text edit boxes 1504 that allow definition of reformatting expressions (e.g., `total = revenue_1 - revenue_2`). Each rule has an

associated output field text edit box 1506 for designating a field that will receive the reformatted result.

`outputspec` – Presents a prompt for an output dataset specification. The displayed control includes a dropdown control for presenting available format options, and a text edit box for entering the name of a specific instance of the output dataset. The blank ok modifier has no effect for output dataset specifications.

`fpath` *starting-point* – Presents a prompt for a file path. The prompt is essentially a text box, but has a "Browse" button next to it that will cause a popup window to appear for browsing for a file path. If the text box is non-blank, then it will be used as the starting point for the browsing operation; if it is blank, the *starting-point* argument is used.

`rpath` *starting-point* – Presents a prompt for a repository path. The prompt is essentially a text box, but has a "Browse" button next to it that will cause a popup window to appear for browsing. If the text box is non-blank, then it will be used as the starting point for the browsing operation; if it is blank, the starting-point argument is used.

`radiofpath` *choice-list [description-list]* – Like `radioplus`, but presents an fpath-style box-plus-browse-button in the "write-in" slot.

`radiorpath` *choice-list [description-list]* – Like `radioplus`, but presents an rpath-style box-plus-browse-button in the "write-in" slot.

*Conditional Components*

Some implementations include a conditional components mechanism that permits changes to the structure of the components and flows of a graph based on parameter values and computed metadata. Each component of a graph has a condition which controls whether or not that component will appear in the graph at runtime. The condition can be computed directly or indirectly through runtime parameters. Conditional components can be used for a variety of purposes, such as to optimize or specialize graphs. For optimization, an application might omit processing of certain datasets if values from them will not be used, thus allowing the graph to run more efficiently. For specialization, an application might condition the production of several different output

datasets based on the level of detail desired, or allow execution of one of several optional portions of a graph.

FIG. 16A is a block diagram of a first graph in which a MergeJoin component 1600 joins data from files A and B and outputs the result to an output file 1602. FIG. 16B is a block diagram of a second graph in which a Rollup component 1604 aggregates data from file A and outputs the result to an output file 1602. FIG. 16C is a block diagram of a graph in which a MergeJoin component 1606 joins data from files A and B, and a Rollup component 1608 aggregates the resulting data and outputs a final result to an output file 1602. Using conditional components, these three graphs can be combined into a single graph that initially looks like the graph of FIG. 16C, but the exact structure of which is not determined until runtime. By setting appropriate conditions, the Rollup component 1608 can be replaced by a connection (flow), resulting in a runtime graph similar to the graph of FIG. 16A. Similarly, by setting appropriate conditions, the MergeJoin component 1606 can be replaced by a connection (flow) to file A, resulting in a runtime graph similar to the graph of FIG. 16B.

In the illustrated embodiment, a conditional component can be any graph component that defines a vertex (i.e., a dataset component such as an input/output file, a processing component such as a reformat or sort component, or other graphs, known as subgraphs). In the preferred embodiment, a conditional component is controlled by two special parameters: a *Condition and a Condition-interpretation.* A Condition is a boolean expression or value whose evaluation is deferred until runtime. In the illustrated embodiment, the values "false" and "0" specify a false condition, all other values (including empty) indicate a true condition. A Condition-interpretation parameter has two allowed mutually exclusive values: *Remove Completely* and *Replace With Flow*.

FIG. 17 is a diagram of one embodiment of a graphical dialog 1700 presenting a Condition 1702 having a Condition-interpretation control 1704. The Condition-interpretation control 1704 allows selection of either a Remove Completely interpretation 1706 or a *Replace With Flow* interpretation 1708.

*Remove Completely*: With this interpretation, if the Condition is met, the component and all of its connected flows (i.e., graph links or edges) are to be removed from the graph. An active *Remove Completely* condition functionally removes the

component and all its directly connected flows from a graph. *Remove Completely* conditions can be used on any component.

A conditional component that is removed from a graph can "poison" other connected components that depend on the presence of the conditional component, causing their removal. FIG. 18 is a diagram of a graph 1800 showing a situation in which such poisoning arises. If the condition on the Input File component 1802 indicates removal and its corresponding condition-interpretation is *Remove Completely*, then both the Input File component 1802 and its connected flow are removed from the graph 1800. This in turn poisons the Sort component 1804, causing it to be removed because its input is a required input port, but there are no longer any data flows connected to it. This in turn poisons the Rollup component 1806, causing it to be removed because its input is a required input port, but there are no longer any data flows connected to it. The only thing that stops this "poison of disappearance" is connection to an optional or counted port of a downstream component. Thus, the entire sort-rollup graph branch 1808 is effectively removed from the graph 1800 when the condition on the Input File component 1802 indicates removal. The result in FIG. 18 is that the nominally 3-input Join component 1810 of the original graph structure becomes a 2-input Join component at runtime.

In one implementation, the detailed semantics of poisoning (also known as "implied conditions") are as follows:

- If a component has a required port and there are no live flows connected to it, the component and all flows connected to it are removed from the graph.
- If a component is removed completely from a graph, then all flows connected to its ports are removed from the graph.
- If a component is replaced with a flow, then all flows connected to all ports other than that component's designated input port and designated output port are removed from the graph.
- If a required indexed port has no live flows connected to it, then for each corresponding optional indexed port with the same index, any flows connected to that corresponding port are removed from the graph.

There are some surprising consequences of these rules. For example, a component with only optional ports can never be removed because of poisoning. Therefore, it must be explicitly removed if desired.

FIG. 19 is a flowchart that summarizes the process of runtime preparation of a graph that includes a *Remove Completely* conditional component. If the Condition-interpretation is *Remove Completely* and the Condition is not met (STEP 1900), then the conditional COMPONENT is not removed from the graph (STEP 1902). If the Condition is met (Step 1900), then the conditional component is removed from the graph, along with all flows connected to that component (STEP 1904). All "poisoned" components and flows are then removed from the graph, in accordance with the rules set forth above (STEP 1906).

*Replace With Flow*: With this interpretation, if the Condition is met, the component is to be replaced with a flow (i.e., a graph edge). *A Replace With Flow* condition-interpretation needs additional information. Referring to FIG. 17, the user designates an input port 1710 (or a family of counted ports) and an output port 1712 (or a family of counted ports) through which to make connections when the component is removed from a graph. By default, if there is exactly one required input port or counted port, and exactly one required output port or counted port, those are the designated flow-through connection ports (termed the *designated input port and the designated output port*, respectively). A required port is one that requires at least one flow to be connected.

FIG. 20 is a flowchart that summarizes the process of runtime preparation of a graph that includes a *Replace With Flow* conditional component for a particular embodiment of the invention. Because of the dependency of some components on certain available inputs and outputs in the illustrated embodiment (which is based on components available in the CO>OPERATING SYSTEM®), several rules apply to this implementation and use of a *Replace With Flow* condition:

- If the Condition-interpretation is *Replace with Flow* and the Condition is not met (STEP 2000), then the conditional component is not removed from the graph (STEP 2002).

- A component with a designated input port and a designated output port can be replaced with a flow only if there is exactly one live straight flow connected to its designated input port, and exactly one live straight flow connected to its designated output port (a "live" flow is a flow that has not been removed at run-time) (STEP 2004). If so, the component itself is removed from the graph, and the straight live flow connected to its designated input port and the straight live flow connected to its designated output port are linked together (STEP 2006). Any other flows directly linked to the removed component's other ports (i.e., any ports other than the specially designated input and output ports) are removed from the graph. Any "poisoned" components and flows that were connected to the removed component are removed, as described above (STEP 2008).

- If a component with a *Replace With Flow* condition has live flows attached to more than one designated input port in a family of counted inputs (STEP 2010), then it is not removed from a graph, because the component is needed to make the graph valid (STEP 2012).

- Components that have live fan-in-flows on required inputs require special handling. A "live fan-in flow" means either the component has a live fan-in or all-to-all flow connected to a required input port, or it has more than one live straight flow connected to a single required input port. For such components, interpreting a *Replace With Flow* condition should replace the conditional component with a gather component which gathers all of live input flows (STEP 2014). Any "poisoned" flows and components that were connected to the replaced component are then removed, as described above (STEP 2016).

*Aspects of Metadata Propagation*

Metadata for a graph can be supplied, for example, by a graph developer, by a graph user, or by propagation from another portion of the graph. Various kinds of metadata can be propagated, including metadata associated with the data or computations on the data such as: a record format for a port (e.g., sequence of fields and data types of

records flowing into or out of a port), sortedness, compression method, character set, binary representation (big-endian, little-endian), partitioning, what computing resources (e.g., processor, temporary disk space) the component may use, data transformations, and amounts of memory the component may use. Various aspects of graph construction can affect the propagation of metadata. Two of these aspects are described below.

*Propagation After Component Removal*

In some implementations, when a flow is generated after the removal of a graph component, a choice must be made as to how metadata defining the data in such flow should propagate in the revised graph. Metadata may be available from either end of the flow. In some implementations, the metadata from the upstream end of the flow is preferred.

If the upstream end of the flow is a removed component (or a component that has been replaced by a gather component), then the GDE 802 finds metadata for the flow by "walking" upstream in the graph until it finds a component that has not been removed. The metadata exposed by that upstream component is used to define the characteristics of the data for the generated flow.

*Propagation of Transformed Metadata*

As described above, metadata can propagate even over transforming internal data paths by allowing metadata associated with a port to be specified as a function of one or more parameters, including metadata for another port. For example, FIG. 30A shows a graph 3000 that computes a join operation on data from data set 3002 and data set 3004. In this example, a graph developer supplies metadata at output ports of the data sets. This metadata is then propagated to a "smart join" component 3006 that computes a join operation on the records of the input data sets. For example, metadata propagates from output port 3008 to input port 3010. The metadata is then transformed by the "smart join" component 3006 and propagated to an input port 3017 of a filter component 3018 from an output port 3016 of the "smart join" component 3006.

FIG. 30B shows a sub-graph implementing the "smart join" component 3006. The component 3006 uses a *key_field* parameter whose value represents the key field of the

join operation performed by a join component 3050. The component 3006 also uses the *key_field* parameter as a condition for including conditional sort components 3054 and 3056. If the records flowing into the input port 3010 are already sorted on the *key_field*, then the sort component 3054 is conditioned out. Similarly, if the records flowing into the input port 3014 are already sorted on the *key_field*, then the sort component 3056 is conditioned out. If either flow of input records are not already sorted on the *key_field*, then the sort components 3054 and 3056 sort the records before they flow into the join component 3050.

To enable propagation of transformed metadata through this "smart join" component, a graph developer defines the metadata (e.g., metadata for describing the fields) for the output port 3016 of the "smart join" component 3006 as a function of metadata for the first input port 3010 *input0.metadata*, metadata for the second input port 3014 input1.metadata, and the key field parameter *key_field*:

*output.metadata = metadata_join(key_field, input0.metadata, input1.metadata)*

The output port metadata is determined by binding the function arguments to values (with respect to the appropriate context) and performing the function *metadata_join* on the results. In this example, since metadata for the ports 3010 and 3014 are undefined, propagated metadata are bound to the metadata parameters *input0.metadata and input1.metadata.* A user supplies metadata for the output port 3008 that specifies fields "A" and "B" for records flowing from port 3008 to input port 3010 of the "smart join" component 3006. The user also supplies metadata for the output port 3012 that specifies fields "A" and "C" for records flowing from port 3012 to input port 3014 of the "smart join" component 3306. This user-supplied metadata propagates to the ports 3010 and 3014. The key field for the join operation is field A, so the "formal parameter" *key_field* is bound to the value "A."

The function *metadata_join* determines the output metadata by first determining whether the value of the *key_field* parameter is a member of both sets of fields specified *by input0.metadata and input1.metadata.* If so, the output metadata is the union of the two sets of fields. If not, the output metadata indicates an empty set of fields.

After the metadata propagates to the input ports of the "smart join" component 3006 (or is otherwise supplied, for example, by a user), the transformed metadata for the

output port of the "smart join" component 3006 includes fields A, B and C. This transformed metadata can then be propagated to other components. In this example, the transformed metadata propagates to the filter component 3018.

Metadata, whether supplied by a user or propagated between ports, can be displayed to the user. For example, the user can use an input device (e.g., a mouse) to select a portion of a component for which to view metadata values. The metadata propagation can also be triggered in response to such a user selection.

*Exemplary metadata propagation process*

FIG. 31 shows a flowchart for an exemplary metadata propagation process 3100. The process 3100 can be performed, for example, each time there is a change in a graph, in response to a user action, and/or just before the graph is run. The process 3100 generates 3102 a worklist with each component in the graph ordered according to the partial ordering determined by the flows (e.g., component A comes before component B if there is a flow from component A to component B). Where flows do not determine a unique order between two components, alphabetic order of component labels may be used as a tie-breaker. This provides a stable ordering for the components in the worklist (assuming the component labels are unique). If the propagation process 3100 is repeated for a graph (e.g., after the addition of a new component), the new worklist preserves the same order between components previously in the worklist.

The process 3100 starts at the beginning of the worklist and, for each component in the worklist, the process 3100 propagates metadata internally 3104 within the component (e.g., from an input port to an output port, or from an output port to an input port) based on a specification of the sub-graph implementing the component (e.g., an data flow in the sub-graph). This internal metadata propagation includes transferring metadata untransformed between ports on either end of an non-transforming data path. Internal metadata propagation also includes deriving metadata for a port that has a metadata definition that refers to parameters of the graph and/or metadata for other port(s), as described above. When the process 3100 encounters such a metadata definition, the process 3100 evaluates any parameters whose values are needed to derive the metadata.

After performing internal metadata propagation for a component on the worklist, the process 3100 propagates metadata externally 3106 from each port of the component that has metadata to a port of a related component that does not have metadata. Any component that acquires metadata by this external propagation is moved 3108 to the end of the worklist. The process 3100 terminates 3110 after the last component on the worklist is processed.

One type of relationship between components that supports this type of external metadata propagation is a data flow link between ports of two components (e.g., from an input port to an output port, or from an output port to an input port).

Another type of relationship between components that supports this type of external metadata propagation is a link indicating that metadata for one port may also be used for another port. This type of "metadata link" does not necessarily correspond to a data flow link. For example, a port can have a metadata link to metadata in a graph that is not associated with any port in particular.

*Runtime Parameters in Componentized Sub-Graphs*

Before a sub-graph is "componentized" to be used as a component in another graph, various characteristics of the component are defined, such as the input and/or output ports of the component. For a sub-graph that includes components with runtime parameters, a prompting order for the runtime parameters should be selected. Since components in a graph are not necessarily sequentially ordered, there can be multiple possible global orderings of the runtime parameters for prompting a user. Some of the global orderings are not as consistent with the original orderings associated with each component. It is useful to generate a global ordering for prompting that preserves as much as possible the orderings of the parameters in each component, while reordering when appropriate to take dependencies into account. For example, a component may order a prompt asking "what data would you to process?" before a prompt asking "where would you like to store the processed data?" Even though it may be possible to provide the prompts in either order, it may be desirable to provide the prompts in this order.

Since it may be necessary to evaluate non-prompted runtime parameters in the process of evaluating prompted runtime parameters, the prompting order is obtained from

an evaluation order for all of the runtime parameters. One approach for determining an evaluation order for the runtime parameters of a graph (including parameters for the graph that are not associated with any component) includes performing a topological sort based on one or more directed acyclic graphs representing dependencies among the parameters. However, some topological sort algorithms may reorder parameters unnecessarily, resulting in an undesirable prompting order for runtime parameters.

*Sorting Example 1*

In a first example, a parameter sorting process provides an initial list of parameters for parameters of two graph components: Component I, and Component II connected to Component I. In this example, the parameters have only "intra-component" dependencies. That is, parameters of a component depend only on other parameters in the same component. The parameters are defined as follows.

Component I includes the following parameters:

```
x = ${prompt_for "text"}
y = x + ${prompt_for "text"}
     z = x + y + ${prompt_for "text"}
q = ${prompt_for "text"}
```

Component II includes the following parameters:

```
a = ${prompt_for "text"}
b = a + ${prompt_for "text"}
     c = ${prompt_for "text"}
```

The order in which the parameters are listed define a desired order in which to prompt a user for values. The initial list of parameters maintains this "initial ordering" for each component. An "ordinal" is assigned to each parameter to indicate that parameter's place in the initial ordering. The following table lists the parameters in this initial ordering.

| Parameter | Ordinal | Dependencies |
|---|---|---|
| X | 0 | |
| Y | 1 | x |
| Z | 2 | x, y |
| Q | 3 | |
| A | 4 | |
| B | 5 | a |
| C | 6 | |

The "dependencies" column indicates other parameters on which the listed parameter depends. The dependencies impose an ordering constraint on the evaluation of the parameters: a parameter needs to be defined before it is used (e.g., referenced) by another parameter.

A "common topological sort" algorithm passes through the list transferring parameters with zero dependencies into an ordered output list on each pass. After each pass, any transferred parameters are removed from the dependencies column. This process repeats until all parameters have been transferred. The order of parameters in the output list represents the "final ordering" such that parameters that depend on other parameters are evaluated after those other parameters have been evaluated.

In this example, on the first pass, the parameters x, q, a and c are transferred into the output list. On the second pass, the parameters y and b are transferred into the output list. On the third and final pass, parameter z is transferred into the output list. Thus, the final ordering for the parameters is: x, q, a, c, y, b, z. While this ordering does satisfy the ordering constraint imposed by the parameter dependencies, it unnecessarily reorders the parameters. In this example, the initial ordering also satisfies the ordering constraint imposed by the parameter dependencies.

Other approaches for determining an evaluation order for the parameters of a graph that satisfies the ordering constraint do respect the initial ordering. For example, some approaches order the parameters to satisfy the ordering constraint, choosing the ordering according to a criterion based on the initial ordering. The criterion can include

any of a variety of criteria that give preference to keeping the order close to the initial ordering (e.g., minimize a metric based on changes to the initial ordering). In some cases, there may not be a unique "best" ordering, since multiple orderings may satisfy a given criterion equally well according to the criterion.

An example of an approach that respects the initial ordering is a "modified topological sort" approach. In this approach, the criterion based on the initial ordering is to minimize the number of parameters that are transferred from the initial list before a preceding parameter that does not depend on any untransferred parameter is transferred. In other words, the "modified topological sort" removes a transferred parameter from the dependencies column before transferring the next parameter with zero dependencies. For the example above, the "modified topological sort" approach generates a final ordering that is the same as the initial ordering: x, y, z, q, a, b, c.

*Modified Topological Sort Process Respecting Initial Ordering*

Pseudocode is given below for two exemplary "modified topological sort" processes that both respect initial ordering as determined by an assigned ordinal for each parameter. The second process includes an optimization to improve time efficiency for some cases. The processes manipulate data structures generated from input data for the parameters.

Assuming there are N parameters to be ordered, the input data includes a list of N triples consisting of a unique parameter name, a set of parameters upon which the named parameter depends (called a "dependency set") and an optional attribute data object storing information related to the named parameter.

Associated with this input data are one or more directed acyclic graphs that represent the dependencies among the parameters, called "dependency graphs." Each unique parameter name corresponds to a node in a dependency graph, and the associated dependency set corresponds to a set of links from other nodes to that node. So a link points from a first node for a first parameter to a second node for a second parameter that depends on the first parameter. Alternatively, the correspondence between the link direction and parameter dependency could be reversed.

An output data structure *result_list* includes a list of the *N* parameters from the input data reordered (if necessary) so that a parameter is evaluated before it is used for evaluating another parameter while giving preference to keeping the order close to the initial ordering. To generate the output data structure *result_list*, the processes "eliminate" parameters by transferring parameters one at a time from a working data structure *param_list* to the output data structure *result_list*. The output data structure is complete after all parameters have been eliminated.

A first "modified topological sort" process includes two phases. In the first phase, the process builds working data structures based on the input data for use in generating the sorted output data structure. In the second phase, the process iteratively sorts and eliminates parameters according to the dependency constraint represented by these working data structures.

Some of the working data structures that the process builds in the first phase are dictionaries, which are data structures based on hashing. Items in dictionaries can be accessed effectively in O(log*N*) time. The following exemplary data structures are built in the first phase:

*parm_list[index]*: an ordered list of non-eliminated parameter names, indexed by a number index (where index = 0 corresponds to the first item in the list). This data structure is "dynamic" (i.e., changes during the execution of the process). The list is indexed by position, such that if an item is removed from the middle of the list, then the index of items after the removed item are shifted accordingly.

*n_dependencies_dict[name]*: a dictionary keyed by a parameter name *(name),* whose entries contain the number of parameters on which the keyed parameter depends. This dictionary is dynamic.

*dependers_dict[name]:* a dictionary keyed by a parameter name *(name),* whose entries are dictionaries (also keyed by parameter name), representing the set of parameters that depend on the keyed parameter. This dictionary is "static" (i.e., does not change during execution of the process).

*order_dict[name]*: a dictionary keyed by a parameter name *(name)*, storing the ordinal position, an integer ranging from 0 to N-1, of the parameter in the initial ordering. This dictionary is static.

*attribute_dict[name]*: a dictionary keyed by a parameter name *(name),* storing the optional attribute data object for the keyed parameter. This dictionary is static.

*result_list[index]*: an ordered list of parameter names and attributes representing the output of the process, indexed by a number *index* (where *index* = 0 corresponds to the first item in the list). This data structure is initially empty. This data structure is dynamic.

For the purposes of analyzing the time efficiency of the processes, the average "degree" (or number of links from a node) of the dependency graphs is assumed to be z. Building these data structures take O(N) time, except for *n_dependencies_dict* and *dependers_dict,* which take O(N*z) time.

In the second phase, the process sorts the parameters in the *param_list* data structure according to a sort criterion *by_n_deps_and_order* that orders parameters first by the number of non-eliminated parameters on which they depend (i.e., by their value of *n_dependencies_dict*), from lowest to highest, and then by their ordinal (i.e., by their value of *order_dict*), from lowest to highest. The process then eliminates the first parameter in the sorted *param_list*. The value of *n_dependencies_dict* for this parameter should be zero. (If the value of *n_dependencies_dict* for the first parameter in the sorted *param_list* is not zero, then an error is flagged.)

To eliminate a parameter, the process appends it to *result_list* (along with any corresponding attributes) and decrements the dependency count (i.e., the value of *n_dependencies_dict*) of all of its dependers (i.e., parameters in *dependers_dict*) by one. Finally, the parameter is deleted from *parm_list*. This sorting and eliminating of the resulting first parameter is repeated until all parameters have been eliminated.

The following is a pseudocode definition for an eliminate procedure:

```
def eliminate(list, index):
    result_list.append( (list[index], attribute_dict[list[index]]))
    for depender in dependers_dict[list[index]]:
        n_dependencies_dict[depender] = n_dependencies_dict[depender] - 1
    delete list[index]
```

The arguments of the *eliminate* procedure are list (whose value is, e.g., *param_list*) and index. The function *result_list.append* appends the indicated list item at position index along with its associated attribute to *result_list*. Then, the procedure

decrements the value of *n_dependencies_dict* for each parameter *depender* that is a member of the *dependers_dict* data structure, keyed on the parameter being eliminated. Then, the procedure deletes the parameter from *list*. The run time for the *eliminate* procedure is O(zlogN).

The following is pseudocode for a sort/eliminate loop for the first "modified topological sort" process:

```
while parm_list is not empty:
    parm_list.sort(by_n_deps_and_order)
    while parm_list is not empty and n_dependencies_dict[parm_list[0]] == 0:
        eliminate(parm_list, 0)
        parm_list.sort(by_n_deps_and_order)
    if parm_list is not empty and n_dependencies_dict[parm_list[0]] > 0:
        delete parm_list[0]
        < record a circularity error and continue >
```

The process first performs an initial sorting of *param_list* using the function *parm_list.sort(by_n_deps_and_order)* that orders parameters of *param_list* according to the sort criterion *by_n_deps_and_order* described above. The process then performs the eliminate procedure followed by another sorting of *param_list* until *param_list* is empty. The process checks to make sure that the number of dependencies for the first parameter (with index = 0) in *param_list* is zero. If not, the process removes the parameter, records a circularity error, and continues. The sort takes O($N\backslash logN$) and the loop range is N, so the estimate for the overall run time for the loop is O($N$2log $N$).

A second "modified topological sort" process takes advantage of the cases in which the dependency graphs are sparse, such that $z << N$. After one initial sort, the process can maintain the sortedness of a list candidates of parameters that do not depend on any other parameters. This reduces this expected run time as described below.

The following is pseudocode for the second "modified topological sort" process:

```
parm_list.sort(by_n_deps_and_order)
while parm_list is not empty:
    # section 1
    candidates = []
```

```
for p in parm_list:
    if n_dependencies_dict[p] == 0:
        candidates.append(p)
# section 2
while candidates is not empty and n_dependencies_dict[candidates[0]] == 0:
    this_parm = candidates[0]
    eliminate(candidates, 0)
    idx = parm_list.index(this_parm)
    delete parm_list[idx]
    tmp = get_new(this_parm)
    candidates = merge(candidates, tmp)
# section 3
if parm_list is not empty:
    parm_list.sort(by_n_deps_and_order)
    if n_dependencies_dict[parm_list[0]] > 0:
        delete parm_list[0]
        < record a circularity error and continue >
```

The process first performs an initial sorting of *param_list* using the function *parm_list.sort(by_n_deps_and_order)* that orders parameters of *param_list* according to the sort criterion *by_n_deps_and_order* described above. The process then performs a loop having three sections (labeled "# section 1," "# section 2," and "# section 3").

In section 1, the process builds a candidates list that contains only parameters with zero dependencies. The process scans all of the parameters in *parm_list* and appends them to candidates, preserving their relative ordering.

In section 2, the process performs a loop in which parameters from candidates are eliminated and new parameters are merged into *candidates*. The first parameter in *candidates*, saved as *this_parm*, is eliminated from *candidates* and deleted from *param_list*. A function *get_new(this_parm)* returns a list of names of parameters that are members of *dependers_dict* for the newly eliminated *this_parm* and have zero dependencies left. These parameters, representing parameters that have had their last dependency removed, are then sorted according to *by_n_deps_and_order* (to ensure they

ordered according to their respective ordinals) and merged into *candidates*. Thus, the *candidates* list remains a list of zero-dependency parameters sorted by ordinal.

Section 3 is only entered if there is a "circularity error" caused, for example, when two parameters are defined in terms of each other. In this case, the process sorts *parm_list* again, and if the first parameter in *parm_list* has nonzero dependencies it is deleted and the loop repeats with section 1.

Assuming there are no circularity errors, the *N*-parameter list *parm_list* is sorted only at the beginning, resulting in a sorting time of O(*N*log*N*). Thereafter, sorting only occurs on the much smaller list of newly generated zero-dependency parameters resulting from eliminating the parameter at the head of the candidates list. The size of this list is less than z (on average), resulting in a sorting time of O(*z*log*z*) and a merging time of O(*z*). Thus, one iteration of the loop is O(z log z) and the overall time is O( $Nz$ log z + $N$ log $N$ ). For the cases in which *z* does not grow with increasing N, this time is effectively O(*N*log*N*).

*Sorting Example 2*

In another example, a parameter sorting process (e.g., the first or second "modified topological sort" process) determines an initial list of runtime parameters for a graph 3200 having graph components 3202, 3204 and 3206, as shown in FIG. 32A. The graph 3200 also has runtime parameters associated with an output port 3208 of an input data set 3210 and an input port 3212 of an output data set 3214. In this example, the parameters have both "intra-component" dependencies and "inter-component" dependencies. That is, parameters of a component depend on parameters in the same component and parameters in other components. In this example, the inter-component dependencies come about due to flows between components that enable propagation of metadata upon which some parameters depend.

Dependencies are indicated in FIG. 32A by a dotted arrow from a first parameter or port to a second parameter or port. An arrow to a port indicates that the value of the linked parameter propagates from that port to a downstream port. An arrow from a port indicates that a value is propagated to the linked parameter from an upstream port. An

arrow from a first parameter to a second parameter indicates that the value of the second parameter depends on (e.g., references) the value of the first parameter.

FIG. 32B shows a dependency graph 3250 that represents an ordering constraint among parameters p0, p1, p2, p4, p5 and p6 based on the graph 3200. FIG. 32C shows a dependency graph 3252 that represents an ordering constraint among parameters p3, p7, p8 and p9 based on the graph 3200.

The parameter sorting process assigns an ordinal to each of ten parameters p0, p2,…, p9 for various graph elements according to the order of placement of the elements in the graph 3200. In FIG. 32A, the first graph element added to the graph 3200 (e.g., by a user using the GDE 802) is component 3202 having parameter p0, p1 and p2. The second element added is component 3206 having parameters p3, p4 and p5. The third element added is data set 3210 having parameter p6. The fourth element added is data set 3214 having parameter p7. The last element added is data set 3216 having no runtime parameters. The following table lists the parameters in the initial ordering defined by the assigned ordinals.

| Parameter | Ordinal | Dependencies |
|---|---|---|
| p0 | 0 | |
| p1 | 1 | p0, p6 |
| p2 | 2 | p6 |
| p3 | 3 | p8 |
| p4 | 4 | p1 |
| p5 | 5 | p1 |
| p6 | 6 | |
| p7 | 7 | p3 |
| p8 | 8 | |
| p9 | 9 | p8 |

The following listings of the parameters in *param_list* and *result_list* at various stages of processing correspond to the first "modified topological sort" process described above. The *param_list* is shown sorted according to the sort criterion *by_n_deps_and_order* at each stage.

| param_list | result_list |
|---|---|
| p0 p6 p8 p2 p3 p4 p5 p7 p9 p1 | empty |
| p6 p8 p1 p2 p3 p4 p5 p7 p9 | p0 |
| p1 p2 p8 p3 p4 p5 p7 p9 | p0 p6 |
| p2 p4 p5 p8 p3 p7 p9 | p0 p6 p1 |
| p4 p5 p8 p3 p7 p9 | p0 p6 p1 p2 |
| p5 p8 p3 p7 p9 | p0 p6 p1 p2 p4 |
| p8 p3 p7 p9 | p0 p6 p1 p2 p4 p5 |
| p3 p9 p7 | p0 p6 p1 p2 p4 p5 p8 |
| p7 p9 | p0 p6 p1 p2 p4 p5 p8 p3 |
| p9 | p0 p6 p1 p2 p4 p5 p8 p3 p7 |
| *empty* | p0 p6 p1 p2 p4 p5 p8 p3 p7 p9 |

The following listings of the parameters in candidates and *result_list* at various stages of processing correspond to the second "modified topological sort" process described above. It is not necessary to sort candidates between stages since the parameters remain in the same order at each stage.

| candidates | *result_list* |
|---|---|
| p0 p6 p8 | *empty* |
| p6 p8 | p0 |
| p1 p2 p8 | p0 p6 |
| p2 p4 p5 p8 | p0 p6 p1 |
| p4 p5 p8 | p0 p6 p1 p2 |
| p5 p8 | p0 p6 p1 p2 p4 |
| p8 | p0 p6 p1 p2 p4 p5 |
| p3 p9 | p0 p6 p1 p2 p4 p5 p8 |
| p7 p9 | p0 p6 p1 p2 p4 p5 p8 p3 |
| p9 | p0 p6 p1 p2 p4 p5 p8 p3 p7 |

empty p0 p6 p1 p2 p4 p5 p8 p3 p7 p9

Thus, referring to FIG. 33, the "modified topological sort" process 3300, takes as input a desired first ordering 3302 in which to prompt a user for values of runtime parameters, and an ordering constraint 3304 for the parameters (e.g., dependency graphs 3350 and 3352). The process 3300, provides the new ordering 3306 of the set of parameters that satisfies the ordering constraint according to the desired first ordering 3302.

*Typical Usage*

Typically, a user sits in front of the Web Interface 808 and finds in the repository 104 the graph of an application the user would like to run. By scanning all of the objects associated with the application graph, the Web Interface 808 generates web page forms that allow the user to specify values for the runtime parameters of the application. Once all runtime parameters have been specified, the combination of the application and the parameter settings are brought together as a job, which is scheduled for execution by the executive 810. When it comes time to run the job, the executive 810 queues the application for execution under the parallel operating system 806, in known fashion. The parallel operating system 806 collects tracking information and job status and stores this information in the repository 804 so that users and administrators can track the progress and performance of jobs.

*Examples*

FIG. 21 is a diagram of a graph 2100 representing a rollup application without runtime parameters. This graph computes the number of accounts of each kind and writes the results to an output file. Every aspect of this application has been determined by the developer who created the graph: the name of the input file component 2102, the format of the input data, the key and transform rules used to roll up the data in a HashRollup component 2104, the output format, and the name of the output file component 2106. A user can only execute this graph exactly as defined.

FIG. 22 is a diagram of a graph 2200 representing a runtime parameterized version of the rollup application of FIG. 21. The dataflow graph structure of this application is very similar to the non-runtime parameterized version, but the application is much more flexible. Through runtime parameters, an end user may specify the name of the abstracted input dataset 2202 (a reposited object from which the input file name and format will be derived), the rollup key and rollup rules for the HashRollup component 2204, and the name of the output file component 2206.

FIG. 23 is a diagram of one embodiment of a graphical dialog representing a runtime parameters grid 2300 for the example application of FIG. 22. This is a filled in version of the parameters grid shown in FIG. 9. Note that a number of default parameters are defined using the `prompt_for` pseudo-function, as described above, and thus require user input through the Web Interface 808. While the appearance of this graph differs little from the non-runtime parameterized application graph, one or more parameter grids (or other suitable control) enable a developer to completely track all parameters that control the execution of the graph.

FIG. 24A is a diagram of one embodiment of a graphical dialog representing a form 2400 generated by the Web Interface 808 from the information in the parameters grid 2300 of FIG. 23. In this example, the form 2400 presents four runtime parameters for user input: an input dataset repository path 2402, a rollup key 2404, rollup rules 2406, and an output path 2408. FIG. 17B is a diagram of the form 2400 of FIG. 24A filled in by a user with parameter values. Using direct entry and/or edit or browser control buttons associated with the runtime parameters 2402-2408, a user provides corresponding parameter values 2410-2416 for executing the associated graph.

FIG. 25 is a diagram of a graph 2500 representing a runtime parameterized rollup and join application. FIG. 26 is a diagram of one embodiment of a graphical dialog representing a runtime parameters grid 2600 for the example application of FIG. 25. Here, some aspects of the application have been parameterized, but most, including the join key and the input datasets, remain fixed. FIG. 27 is a diagram of one embodiment of a graphical dialog representing a form 2700 generated by the Web Interface 808 from the information in the parameters grid 2600 of FIG. 26. Note that since the input type to the

rollup is known at the time the top-level form is displayed, the rollup rules 2702 can be prompted for in-place.

FIG. 28 is a diagram of a graph 2800 representing a runtime parameterized rollup-join-sort application. While similar to the example in FIG. 25, a conditional sort component 2802 has been added to the graph 2800. FIG. 29 is a diagram of one embodiment of a graphical dialog representing a runtime parameters grid 2900 for the example application shown in FIG 28. The `sort_key` runtime parameter 2902 is prompted for only if the user indicates that sorting is desired. To get this effect, a develop puts a `prompt_for` pseudo-function within an if conditional test for the default value 2904 of the `sort_key` runtime parameter 2902. The if conditional test references a second runtime parameter, `do_sort` 2906. The default value field 2908 and description field 2910 of the `do_sort` parameter 2906 are defined to generate a radio prompt asking the user for a true/false or yes/no answer to the text prompt "Should the data be sorted?". If the value provided for the `do_sort` parameter 2906 is "true", the sort component 2802 will be included as part of the graph at runtime. Otherwise, the sort component 2802 will be removed completely from the graph or replaced with flow, depending on its specified condition interpretation.

*Script Implementation*

While the GDE 802 facilitates construction of parameterized graphs, sometimes there are non-graph programs for which one would like to provide a forms-based interface. Using application-level PL and the repository 804, one can parameterize arbitrary shell scripts. For example, the description of an application can be written to a file with a structure similar to the following:

```
application AppName(
 description("One-line Description"),
 comment("Longer description"),

 parameter ParmName1(
 string, kind(keyword), required,
 description("Short prompt for top-level form"),
```

```
        comment("Longer prompt for out-of-line form"),
        default(${prompt_for …})
        ),

        parameter ParmName2(
        type, kind(derived),
        default(PL-expression)
        ),

        . . . more parameters . . .

        script(="scriptname.ksh")
      )
```

*General Computer Implementation*

The invention may be implemented in hardware or software, or a combination of both (e.g., programmable logic arrays). Unless otherwise specified, the algorithms included as part of the invention are not inherently related to any particular computer or other apparatus. In particular, various general purpose machines may be used with programs written in accordance with the teachings herein, or it may be more convenient to construct more specialized apparatus to perform the required method steps. However, preferably, the invention is implemented in one or more computer programs executing on one or more programmable computer systems each comprising at least one processor, at least one data storage system (including volatile and non-volatile memory and/or storage elements), at least one input device or port, and at least one output device or port. The program code is executed on the processors to perform the functions described herein.

Each such program may be implemented in any desired computer language (including machine, assembly, or high level procedural, logical, or object oriented programming languages) to communicate with a computer system. In any case, the language may be a compiled or interpreted language.

Each such computer program is preferably stored on a storage media or device (e.g., solid state, magnetic, or optical media) readable by a general or special purpose programmable computer, for configuring and operating the computer when the storage media or device is read by the computer system to perform the procedures described herein. The inventive system may also be considered to be implemented as a computer-readable storage medium, configured with a computer program, where the storage medium so configured causes a computer system to operate in a specific and predefined manner to perform the functions described herein.

A number of embodiments of the invention have been described. Nevertheless, it will be understood that various modifications may be made without departing from the spirit and scope of the invention. For example, a number of the function steps described above may be performed in a different order without substantially affecting overall processing. For example, STEPS 1102 and 1112 in FIG. 11 may be performed in reverse order. Accordingly, other embodiments are within the scope of the following claims.

Executing Computations Expressed As Graphs

Overview of Computational Substrate

A graph execution system and a graph execution method for executing a computation expressed as a graph, in accordance with the present invention, will often be used in a computational environment or substrate having the following commonly-available facilities: communication channels, data stores, process control and data access methods. For that reason, the graph execution system and the graph execution method will be described in reference to such a reference substrate, although the system and method are not limited to such substrates.

As to communication channels, the reference substrate provides facilities to create and destroy communication channels, and to do so on remote processors if the system or method are to be used in a distributed environment. Such communication channels may serve to transmit data between two processes on the same processor or between two processes on different processors. Several kinds of communication channels may be provided. As to data stores, the reference substrate provides facilities to create data stores, and to do so on remote processors if the system or method are to be used in a distributed

environment. Data stores are memories that preserve data between jobs or between stages of processing. The substrate also provides mechanisms which allow processes to read/write the contents of data stores. In some cases, the substrate may provide mechanisms by which a process may read/write the contents of a data store located on a remote node, but such facilities are not required by the present invention. As to process control, the substrate provides facilities to start a process, to determine when it has finished execution, and to determine whether or not the process terminated normally. The substrate provides facilities to start a process on remote processors if the system or method are to be used in a distributed environment. The operation of starting a process must allow parameters to be passed, including the name of the program to be run and identifiers for any files or communication channels to be accessed. As to data access methods, the substrate provides facilities for data access methods that processes running on the substrate may use.

A useful substrate may be developed as an extension of the substrate provided by any of various versions of the UNIX operating system. The UNIX operating system provides two types of communication channels: named pipes and TCP/IP streams. A named pipe is a point-to-point communication channel that may be used to transmit data between processes on the same node. (The term "node" refers to a computer, possibly having multiple processors and storage devices or data stores, but having a single pool of shared memory.) A named pipe is identified by a path in a file system. A TCP/IP stream is a point-to-point communication channel that may be used to transmit data between two processes anywhere on a TCP/IP network (e.g., the Internet). Establishing a TCP/IP stream requires the use of a protocol by which the two processes establish a connection. Many such protocols are in use, but most are oriented towards establishing client-server connections rather than peer-to-peer connections required by the system.

The extended substrate provides one additional type of channel called shared memory. Shared memory is a pool of memory that is accessible by multiple processes and that may be used for transmitting data, in known fashion. A shared memory channel includes mechanisms for synchronizing access. Shared memory is applicable to transmitting data between processes on the same node and is an efficient substitute for named pipes.

For use with the present invention, the TCP/IP connection protocol of the extended substrate should be suitable for peer-to-peer links. In such a protocol, an entity creates a "stream identifier" and arranges for both end points of the desired stream to receive the stream identifier. The two end points give the stream identifier to the substrate and receive in response a connection to a TCP/IP stream. A similar mechanism is used to establish shared memory communication channels. Creation of such a protocol is well-known in the art.

The UNIX operating system provides three data access methods: a file interface, a stream interface, and a file descriptor interface. With the file interface, a program may be provided the name of a file, and then may open it, read from it, write to it, and seek within it. With a stream interface, a program may be provided the name of a file or a named pipe, and then may open it, read from it, or write to it. A stream interface does not allow the use of a seek operation or other operations that are applicable only to files. With a file descriptor interface, a file descriptor (identified by a number, for example) may be bound to a file, a named pipe, or a TCP/IP stream before a program is invoked. The program may then read from the file descriptor or write to it. This is similar to the stream interface, except that with the stream interface the stream is opened outside the program, that is, before the program is invoked.

For use with the present invention, the extended substrate provides one additional access method, stream object connections (SOCs). A "stream object" is created by presenting the extended substrate with a "stream object identifier" (a unique string), a "communication method name", and a "channel/file identifier", which may be the name of a file or a named pipe, or an identifier to be used in creating a TCP/IP stream or shared memory channel. The caller must also provide the identities of the nodes where the source and destination are located.

The UNIX operating system provides one type of data store: files. A file is sequence of bytes, stored typically on disk. A file is identified by a path in a file system and an identifier identifying its host, i.e., the node on which it resides.

The UNIX operating system provides process control. To start a process, the following information is provided: the name of the program to run, any command line arguments, and any bindings for file descriptors. Command line arguments may contain

the names of files and named pipes to be accessed by the program. Each binding for a file descriptor consists of a file descriptor number and either a path identifying a file or named pipe, or an auxiliary channel identifying a TCP/IP stream. In addition, command line arguments may contain values for environment variables, which are commonly required by UNIX programs to describe where certain resources may be located or to configure execution options.

Overview of Graphs

The inventive system keeps track of the state of the computation in the state variables of a set of process vertices, a set of file vertices, and a set of data links, each of which will now be described. The operation of a system using these constructs will be described later.

To illustrate some of the concepts of the present invention, an example of the prior art will be used. FIG. 34A illustrates the flow of data through a simplified example of a prior art payroll system. The inputs to this example are an old master file 3410 containing permanent records for all employees and an update file 3412 containing one week's worth of time sheets. The old master file 3410 is sorted by employee ID. The graph shows the following flow of data:

(1) The update file 3412 is sorted, step 3414.

(2) The updates are checked, step 3416, with bad records being put into a bad records file 3418.

(3) The updates from step 3416 plus the old master file 3410 are processed, step 3420, to produce a new master file 3422 plus a set of paychecks 3424.

*Process Vertex*

A process vertex contains the following information:

- a process vertex identifier;
- a program template;
- a working directory identifier that identifies a working directory (that is, a directory where scratch files may be created); and

- a working node identifier that identifies a working node (that is, a node where processing is to take place);
- a state variable which has one of the following values: Disabled, Enabled, Runnable, Unrunnable, or Done (initially, Disabled).
- A flag to indicate if the vertex is an adapter vertex (discussed below).

The following table illustrates the contents of three process vertices for the sort, check, and process processes (programs) shown in FIG. 34A:

| Identifier | Program Template | Working Directory | Working Node | State | Adapter Vertex? |
|---|---|---|---|---|---|
| Sort | Sort Template | /work/tmp | node 1 | Disabled | False |
| Check | Check Template | /work/tmp | node 1 | Disabled | False |
| Process | Process Template | /work/tmp | node 1 | Disabled | False |

*File Vertex*

A file vertex relates to a file and contains the following information:

- a file vertex identifier;
- a data node identifier;
- a data file identifier;
- a working node identifier that identifies a working node, which may be used by programs wishing to access the file (this will usually be identical to the data node);
- a working directory identifier that identifies where programs wishing to access the file may create scratch files (this normally will be in the same file system as the data file); and
- a state variable that has one of the following values: Complete, Incomplete (initially, Incomplete).

In cases where the computational substrate provides for data stores other than files, the information in a file vertex may be extended to include an indicator indicating the type of data store and information required to identify and use it.

The following table illustrates the contents of five file vertices for the files shown in the graph of FIG. 34A:

| File Vertex Identifier | Data Node | Data File Identifier | Working Node | Working Directory | State |
|---|---|---|---|---|---|
| OldMaster | node 0 | /data/master.old | node 0 | /work/tmp | Incomplete |
| Update | node 2 | /input/update | node 2 | /work/tmp | Incomplete |
| BadRecs | node 1 | /err/bad-recs | node 1 | /work/tmp | Incomplete |
| Paychecks | node 1 | /output/paychk | node 1 | /work/tmp | Incomplete |
| NewMaster | node 0 | /data/master.new | node 0 | /work/tmp | Incomplete |

*Data Link*

A data link (or, simply, a "link", for short) contains the following information:

- a source vertex - either a file vertex or a process vertex;
- a source port identifier - if the source is a file vertex, only Output is permitted;
- a destination vertex - either a file vertex or a process vertex;
- a destination port identifier - if the destination is a file vertex, only Input is permitted;
- a communication method identifier, which may be Unbound or the name of a communication channel type such as File, NamedPipe, TCP/IP, or Shared Memory (initially, Unbound); and
- a state variable that may be either Unlaunched or Launched (initially, Unlaunched).

More than one data link may be attached to a given port of a given file vertex or process vertex.

In FIG. 34A, the connecting arrows graphically depict linkages between process and file vertices. The concept of a link is given more concrete form in the present invention. Turning to FIG. 34B, graph 3430 illustrates process vertices, file vertices, and data links derived in accordance with the present invention from the graph 3400 illustrated in FIG. 34A.

The following table tabulates the seven data links 3432, 3434, 3436, 3438, 3440, 3442, and 3444 shown in graph 3430:

| Ref. Num. | Source Vertex | Source Port ID | Destin. Vertex | Destin. Port ID | Comm. Method | State |
|---|---|---|---|---|---|---|
| 132 | Update | Output | Sort | Input | Unbound | Unlaunched |
| 134 | Sort | Output | Check | Input | Unbound | Unlaunched |
| 136 | Check | Bad | BadRecs | Input | Unbound | Unlaunched |
| 138 | Check | Good | Process | Update | Unbound | Unlaunched |
| 140 | OldMaster | Output | Process | Master.in | Unbound | Unlaunched |
| 142 | Process | Master.out | NewMaster | Input | Unbound | Unlaunched |
| 144 | Process | Paychecks | Paychecks | Input | Unbound | Unlaunched |

*Program Template*

A process vertex includes a program template. A program template contains two basic kinds of information about a program: (1) information needed to invoke the program, such as the name of the program, command line arguments, and environment variables; and (2) an array of port descriptors that describe the means by which the program accesses data. The exact form of the invocation information depends on the computational substrate. For the UNIX-based substrate that has been described, the program name is the name of an executable file plus command line arguments consisting of a series of strings separated by spaces. Some of these strings may be in the form of "$portname", where "portname" is the name of one of the ports in the template, and where that port must use the file interface or stream interface data access method. In this case, the string "$portname" will be replaced, at invocation time, by the names of the files or named pipes connected to that port.

The following table illustrates the contents of illustrative program templates for the three programs (processes) shown in the graph 3430 of FIG. 34B:

| Template Name | Program | Arguments | Port ID | Direction | Method |
|---|---|---|---|---|---|
| Sort Template | /bin/sort | $Input<br>EmpNo | Input<br>Output | input<br>output | ReqsFile<br>ReqsFD |
| Check Template | /pay/check | $Input<br>$Good<br>$Bad | Input<br>Good<br>Bad | input<br>output<br>output | ReqsNamedPipe<br>ReqsNamedPipe<br>ReqsNamedPipe |
| Procedure Template | /pay/process | null | Master.in<br>Update<br>Master.out<br>Paychecks | input<br>input<br>output<br>output | ReqsSOC<br>ReqsSOC<br>ReqsSOC<br>ReqsSOC |

The information needed to invoke the program may be provided in two forms. First, it may be explicitly stored in the program templates as illustrated in the table above. Second, this information may be dynamically generated, in which case addresses of routines that generate the information will be stored in the program template.

A port descriptor in a program template contains the following information shown as the right-most three elements in the table above:

- a port identifier for the port;
- an indication as to whether the port is used for input or output; and
- an acceptable-method code, such as ReqsFile, ReqsNamedPipe, ReqsFD, or ReqsSOC ("Reqs" stands for "Requires"), that indicates which communication methods are acceptable to the port.

These acceptable-method codes refer to the data access methods supported by the computation substrate.

**Driver Program**

Turning to FIG. 35, a driver program 3500 (or, simply, a "driver", for short) provides a means for depicting a graph, based on input from a user 3502 received through a user interface 3504. One or more graph data structures 3506 representing a particular graph (for example, such as is shown in FIG. 34B) are generated by the driver 3500. A graph execution control function 3508 of the driver 3500 then permits the driver 3500 to perform the following external control 3510 functions in any order and as often as required in order to interact with external processes 3512:

- create a process vertex;
- create a file vertex;
- create a data link between any pair of vertices (of either kind);
- change the current state of a process from Disabled to Enabled; and
- cause the graph to be executed.

The driver 3500 performs these operations ultimately in response to requests made by a user 3502 and using information provided by a user 3502 through the user interface 3504. A user 3502 may be a person providing input directly through a graphical user interface to the driver 3500. Alternatively, a user 3502 may be a separate program that controls the driver 3500 through, for example, an object oriented or procedural interface. In this way, the user 3502 may use the driver 3500 to build and to control the execution of a graph.

The driver 3500 may create a process vertex for a graph when the user 3502 has provided an identifier, a program template, a working node, and a working directory. The working directory may be given a default value based on the working node.

The driver 3500 may create a file vertex for a graph when the user 3502 has provided an identifier, a data node, a data filename, a working node, and a working filename. The working node defaults to the data node in the preferred embodiment. The working filename may be given a default value based on the working node and/or the data filename.

The driver 3500 may create a data link between any pair of vertices of a graph when the user 3502 has provided a source vertex, a destination vertex, a source port

identifier, and a destination port identifier. In the preferred embodiment, the source port identifier defaults to Output and the designation port identifier defaults to Input.

By enabling a user 3502 to control the Enabled/Disabled state of process vertices, the driver 3500 enables the user 3502 exercise control over the order of execution of processing, which is described below, by selectively enabling subsets of the process vertices.

**Executing a Graph**

Turning to FIG. 36, after an initial graph is generated, the driver 3500 controls execution of the graph, and hence the processes depicted by the graph. When the driver 3500 executes a graph, it does so by performing the following general phases A-I:

A. As long as any one of the process vertices is in the Enabled state, the driver 3500 repeats the following steps B-I. The driver 3500 may sometimes omit phases C, D, and I, and may intermingle the operations performed in steps B, C, E, and H.

B. The driver 3500 prepares the graph for execution. In this phase, the driver 3500 identifies Runnable process vertices, chooses communication methods for links, and may generate adapter nodes, as will be described.

C. The driver 3500 launches data links, as will be described later in more detail. In this phase, the driver 3500 creates certain computational structures required to implement communication methods, as will be described.

D. The driver 3500 creates any other data structures or files required by the computational substrate. For the extended substrate described above, the driver 3500 creates a link file, as will be described. This permits programs to access graph connectivity information at run time.

E. The driver 3500 launches processes, as will be described.

F. The driver 3500 waits for the processes to terminate. This phase completes when all processes have terminated successfully, or when any process terminates abnormally.

G. If any process terminates abnormally, execution of the graph is aborted.

H. Otherwise, all process vertices in the Runnable state transition to the Done state. If no process vertices were in the Runnable state, then cleanup phase I will be performed and control returned to the caller (the user 3502 of the driver 3500, for example) with an indication that execution stalled.

I. The driver 3500 cleans up data links and the link file, as will be described. This cleans up some of the data structures created in phases C and D.

Further details of particular phases are described below.

*Phase B: Preparing a Graph for Execution*

Turning to FIG. 37A, the driver 3500 accesses a graph initially depicted by a user 202 and prepares that graph for execution by applying graph transformations (step 3700). In performing these transformations, the graph data structures defining the initial graph are traversed, in known fashion, to fetch each vertex and any associated links. In the preferred embodiment, five graph transformations are used on the fetched data structures to prepare the graph for execution.

While a graph is still not in executable form (step 3702), the five graph transformations described below may be selected and applied in any order (step 3704) and as often as required (including not at all) until an executable graph is obtained (step 3716). The five graph transformations are (1) inserting a file adapter (step 3706), (2) inserting a communication adapter (step 3708), (3) setting a file vertex's state to Complete (step 3710), (4) setting a process vertex's state to Runnable or Unrunnable (step 3712), and (5) setting a data link's communication method (step 3714). Each of these transformations and the conditions under which each may be performed will now be described.

**Inserting a file adapter**. In this transformation, the driver 3500 replaces a link with a file adapter (that is, with a link, a file vertex, and another link). That is, as each graph data structure representing a link is fetched or accessed during a traverse of the graph data structures 3506 (FIG. 35), a new data structure may be created that modifies, expands on, or substitutes for the original data structure.

For a source (destination) file adapter, the file vertex's host is the same as the source (destination) vertex's host, and the file vertex's file is a new file located in the source (destination) vertex's working directory. This transformation may only be performed if:

(1) the source is either a file vertex or a process vertex which is not in the Done state; and

(2) the destination is either a file vertex in the Incomplete state or a process vertex which is not in the Done state.

**Inserting a communication adapter**. In this transformation, the driver 3500 replaces a link with a communication adapter (that is, with a link, a process vertex, and another link). The process vertex runs a copy program, which copies data from its input to its output, and which can read from/write to any of the communication channels or data stores supported by the underlying substrate. For a source (destination) communication adapter, the process vertex's host is the same as the source (destination) vertex's host, and the working directory is the same as the source (destination) vertex's working directory. The process vertex is created in the Enabled state. This transformation may only be performed if:

(1) the source is either a process vertex in a state other than Done, or a file vertex; and

(2) the destination is either a process vertex in a state other than Done, or a file vertex in the Incomplete state.

**Setting a file vertex's state to Complete**. In this transformation, a file vertex's state is set to Complete. This transformation may only be performed if the file vertex's state is Incomplete and all inputs to the file vertex are process vertices in the Done state.

**Setting a process vertex's state to Runnable or Unrunnable**. In this transformation, a process vertex's state is set either to Runnable or to Unrunnable. This transformation may only be performed if the process vertex's state is Enabled.

**Setting a data link's communication method**. In this transformation, a communication method is set for a data link. This transformation may only be performed if the data link's communication method is Unbound.

A graph that has the following three properties is executable:

(1) All process vertices are in one of the following states: Done, Runnable, Unrunnable, or Disabled.

(2) All data links satisfy all of the following criteria:

1) If either the source or destination of a data link is a Runnable process vertex, then the communication method for the data link must be bound to a particular communication method.

2) If the communication method of a data link is anything other than File, then both its source and destination must be process vertices, and if one process vertex is Runnable, then both process vertices must be Runnable.

3) If the communication method of a data link is File, then its source or destination must be a file vertex. If the destination is a Runnable process vertex, then the source must be a Complete file vertex. If the source is a Runnable file vertex, then the destination must be an Incomplete file vertex.

(3) All links bound to a communication method satisfy the constraints inherent in the communication method:

1) The communication method must be compatible with the access methods for its source and destination ports (this may be determined by consulting the program template). In the case of the extended substrate that has been described, all communication methods are compatible with SOC access; all but Shared Memory are compatible with File Descriptor access; NamedPipe and File are compatible with NamedPipe access; and only files are compatible with File access.

2) Some communication methods require that the nodes of the source and destination vertices be identical. For the extended substrate that has been described, this is true for all communication methods other than TCP/IP.

The graph transformations may be applied in any order (e.g., the graph data structures may be traversed repeatedly until all transformations are complete) until an executable graph is obtained. Turning to FIG. 37B, graph transformations are applied in one embodiment in the following steps taken in the following order: (1) insert file adapters (step 3750); (2) replace file-to-file links (step 3752); (3) identify Complete file vertices (step 3754); (4) identify Unrunnable process vertices (step 3756); (5) identify Runnable process vertices (step 3758); (6) set the remaining Enabled vertices to Unrunnable (step 3760); (7) insert more file adapters where conditions are met (step 3762); (8) choose communication methods (step 3764); and (9) insert communication adapters (step 3766). The steps of this embodiment will now be described in more detail.

**(1) Insert File Adapters.** Turning to FIG. 38A, to insert file adapters, the following steps are performed for all links in the graph (step 3800). If the source port of a link has a data access method requiring the use of a file (step 3802) and the destination is not a file on the same node (step 3804), then insert a source file adapter (step 3806). If the destination port of a link has a data access method requiring the use of a file (step 3808) and the source is not a file on the same node (step 3810), then insert a destination file adapter (step 3812). If the destination of a link is a process vertex in the Disabled state (step 3814) and the source is a process vertex in the Enabled state (step 3816), then insert a destination file adapter (step 3812).

**(2) Replace File-to-File Links**. Turning to FIG. 38B, to replace file-to-file links, the following steps are performed for all links in the graph (step 3820). If the source and the destination are both file vertices (step 3822), then insert a source communication adapter (step 3824). (If, in addition, the source and destination are on different nodes, then also insert a destination communication adapter; not shown).

**(3) Identify Complete File Vertices.** Turning to FIG. 38C, to identify Complete file vertices, the following steps are performed for all file vertices in the graph (step 3830). If all upstream vertices are process vertices in the Done state (step 3832), then set its state to Complete (step 3834).

**(4) Identify Unrunnable Process Vertices**. Turning to FIG. 5D, to identify Unrunnable process vertices, the following steps are performed for all links in the graph (step 3840). An "Unrunnability" test is performed (step 3842) as follows: if the source of a link is an Incomplete file vertex and its destination is a process vertex in the Enabled state, set the state of the process vertex to Unrunnable (step 3844); if the source is a process vertex in any state other than Enabled, and the destination is a process vertex in the Enabled state, then mark the destination process vertex as Unrunnable (step 3844). Repeat this testing until no more vertices may be marked as Unrunnable.

**(5) Identify Runnable Process Vertices**. Turning to FIG. 5E, to identify Runnable process vertices, the following steps are performed for all process vertices in the graph (step 3850). A "Runnability" test is performed (step 3852) as follows: if a vertex is in the Enabled state, and all upstream vertices are either Complete file vertices or Runnable process vertices, then set the state of the vertex to Runnable (step 3854). Repeat this testing until no more vertices may be marked as Runnable.

**(6) Set the Remaining Enabled Vertices to Unrunnable.** Turning to FIG. 5F, to set the remaining Enabled vertices to Unrunnable, the following steps are performed for all process vertices in the graph (step 3860). If a vertex is in the Enabled state (step 3862), then set its state to Unrunnable (step 3864).

**(7) Insert More File Adapters.** Turning to FIG. 5G, to insert more file adapters, the following steps are performed for all links in the graph (step 3870). If the source of a link is a Runnable process vertex (step 3872) and the destination is an Unrunnable process vertex (step 3874), then insert a source file adapter (step 3876).

**(8) Choose Communication Methods.** Turning to FIG. 5H, to choose communication methods, the following steps are performed for all links in the graph (step 3880). This step only applies to links which are attached, at either end, to a runnable process, and which are not bound to a communication method. If a link's source (destination) is a file vertex (step 3881), and its destination (source) is a process vertex on the same node, then set the link's communication method to File (step 3882). Otherwise, choose one of the available communication methods, such that all of the constraints of that method are satisfied (steps 3883-3885). For speed, communication methods may be considered in the order Shared Memory, NamedPipe, and TCP/IP. The first method that satisfies the constraints set forth above is selected (step 3886). In the reference substrate, the following rules may be used: First, if a link is attached to a port which accepts SOC connections, then the link will use Shared Memory if the source and destination are on the same node, or TCP/IP if they are on different nodes. Otherwise, if the source and destination are on the same node, a NamedPipe method will be used. In all other cases, no single communication method suffices, and the system will restore to a communication adapter (below).

**(9) Insert Communication Adapters.** If no single communication method is selected in the preceding step of choosing a communication method and all have been tried (step 3883), continue by inserting a source communication adapter and trying to choose communication methods for the two links of the adapter (step 3887). If this fails (step 3888), try replacing the newly inserted source communication adapter with a destination communication adapter (step 3889). If this fails (step 3890), insert both a source and a destination communication adapter, and choose communication methods for three links in the resulting double adapter (step 3891). In the reference substrate, communication adapters are only required if the source and the destination are on different nodes, and the link is connected to either a file vertex or a process vertex not accepting the SOC connection method. In this case, adapters may be chosen as follows:

- If the source is a file vertex, then insert a source communication adapter. The two links in the source communication adapter will use, in turn, the File and the TCP/IP communication methods.

- If the source is a port not accepting the SOC communication method, then insert a source communication adapter. The two links in the source communication adapter will use, in turn, the TCP/IP and File communication methods.
- If the destination is a file vertex, then insert a destination communication adapter. The two links in the adapter will use, in turn, the TCP/IP and File communication methods.
- If the destination is a port not accepting the SOC communication method, then insert a destination communication adapter. The two links in the adapter will use, in turn, the TCP/IP and NamedPipe communication methods.

*Phase C: Launching Data Links*

Turning to FIG. 39, data links are created in the Unlaunched state and must be launched. To launch links, links are scanned (step 3900) to find links that are Unlaunched (step 3902), bound to communication methods (step 3904), and have Runnable sources or destinations (step 3906). For all such links, identifiers are generated that may be used by the various communication methods (step 3908). For the extended substrate described above, identifiers are created as follows. All links have two identifiers: the stream object identifier and the communication channel/file identifier. The stream object identifier is used by the SOC mechanism, and is identical to the name of the link. The channel/file identifier is used to identify the file, named pipe, shared memory region, or TCP/IP connection employed by the link. Additionally, in cases where the process vertex requires the NamedPipe or File communication methods, the channel/file identifier will be made available so that the process vertex, when launched (see below), will be able to attach to the channel/file using the UNIX file system.

After the identifiers are generated, the substrate is called to create a channel or stream object (step 3910). If the communication method is NamedPipe, the substrate is also called to create the named pipe.

*Phase D: Creating the Link File*

The extended substrate maintains, on each node, a "link file" which enumerates the links which have either a source or a destination on that node. Programs may consult

this link file at run-time to determine which links they must access. This is commonly done for programs which use the SOC interface. Accordingly, in the case of the extended substrate, the system must create the link file. This is done as follows: For every node involved in the computation, the driver 3500 identifies Runnable process vertices assigned to that node and, for every link attached to such a vertex, accumulates and stores in a link file the following information:

- the identifier for the vertex;
- the name of the port the link is attached to;
- the identifier for the communication channel; and
- the identifier for the file or named pipe used to transport data, if applicable.

*Phase E: Launching Processes*

Turning to FIG. 40, processes are launched by performing the following steps for all process vertices in the Runnable state (step 4000). First, the program template for the vertex is used to generate invocation information (step 4002). This information includes:

- the name of the program to be invoked;
- command line arguments (command line arguments may contain identifiers for the communication channels and files associated with the links attached to the vertex);
- optionally, values for various environment variables; and
- optionally, the expected "exit code" of the program.

Invocation information may be generated in one of at least two ways: the driver 3500 may pre store such information in a program template; or the driver 3500 may contain routines for dynamically computing such information, and may store the addresses of those routines in the program template.

Next, the identifier of the vertex and the identifier of the link file for the vertex's node are added to the set of environment variables (step 4004). Then an agent running on the indicated node creates a "program invocation process" (step 4006). If the program's template requires that an input or output port be bound to a UNIX file descriptor, the program invocation process is provided with the name of the file, named pipe, or TCP/IP

stream identifier associated with the input or output port, and will open that file or named pipe using the indicated file descriptor. The program invocation process sets up the required environment variables and runs the indicated program using the indicated command line arguments (step 4008).

*Phase F: Waiting*

Once all processes have been launched, the system will monitor their execution, preferably using the same agents as were used to launch them. Periodically, the system (agent) will notice that a process has exited. When this happens, the system (agent) will determine whether the process exited "normally" or "abnormally". In the case of UNIX, this is done via a termination code. The termination code may indicate that the program aborted due to program error, arithmetic exception, invalid memory access, etc. Such cases are always interpreted as "abnormal termination." Alternately, the program may have exited in a controlled fashion, returning an "exit code" (exit codes comprise a subset of the termination codes.) By convention, an exit code of 0 indicates the program terminated normally, with all other codes indicating abnormal termination. As noted above, the program template may alter this interpretation, e.g., declaring that all exit codes are to be interpreted as "normal" termination.

As soon as the system determines that a process has terminated normally, it may optionally enter a "debugging" routine to allow the user to diagnose the reason for the abnormal termination. Once debugging is complete (or skipped), the system will initiate an abort procedure, e.g., killing all processes which are still running, deleting partially written files, etc. The driver program will then exit.

If the system determines that a process terminated normally, it will note this fact and wait for more processes to terminate. When all processes have terminated normally, the system proceeds to cleaning-up phase I.

*Phase I: Cleaning Up*

After execution of all Runnable processes has finished, the driver 3500 performs the following steps. First, the link file for each node is deleted. Second, all links in the Launched state are scanned. If all process vertices attached to a link are in the Done state,

the substrate will be used to destroy any communication channels associated with the link. For the extended substrate, this is done by obtaining the stream object identifier for the link, and commanding the substrate to destroy the indicated stream object. In the case where the communication method is NamedPipe, this will also cause the named pipe to be deleted. In addition, if a file adapter vertex is Complete, and all processes downstream from it are Done, its file will be deleted.

*Inserting Adapters*

Several circumstances have been mentioned in which an adapter is inserted in place of a link. An adapter is a data link, or a file or process vertex (the adapter vertex), and another data link that are inserted in order to synthesize a communication link using a sequence of communication methods. Adapters in which the adapter vertex is a process are referred to as "communication adapters". Adapters in which the adapter vertex is a file are referred to as "file adapters". In general, communication adapters are added when a combination of communication methods (e.g., a named pipe plus a TCP/IP connection) is required to satisfy the constraints imposed by the source and destination of the link. File adapters are added in cases where the source and destination of a link do not run at the same time (e.g., the source is runnable, but the destination is unrunnable or disabled), or where the source or destination of a link can only attach to files. In the case of communication adapters, the adapter vertex specifies a program which copies its input to its output, and which may attach to any sort of communication channel. The adapter links may subsequently be bound to any communication method, subject to the constraints described in steps 3883-3885 "choosing a communication method". The adapter vertex simply copies data from its input to its output. The adapter links may have any convenient communication method, subject to source and destination constraints.

Turning to FIG. 41, to insert a source adapter (step 4100), a new adapter vertex (step 4102) and a new adapter link are created (step 4104). In the case of a source communication adapter, the adapter vertex is a process vertex having the following characteristics:

- The program template specifies a copy program, i.e., one that copies all input data to its output.

- The program template specifies input and output port descriptors capable of attaching to any communication method.
- The new vertex is flagged as an adapter vertex.
- The working directory and working node of the original source vertex are used as the working directory and node of the adapter vertex.

In the case of a source file adapter, the adapter vertex is a file vertex having the following characteristics:

- The file is located on the node used by the source program.
- The file is located in the working directory of the source program.

  The file is marked as being an "adapter" so that, when it has been consumed by the destination process, it may be deleted.

The new adapter link (step 4104) has the following characteristics:

- The source vertex of the adapter link is the same as the source vertex of the original link, and the source port name of the adapter link is the same as the source port name of the original link.
- The destination vertex of the adapter link is the adapter vertex, and the destination port name of the adapter link is Input.
- The communication method of the adapter link is set to a value specified by the procedure inserting the source adapter (often this will be Unbound).

Lastly, the source of the original data link will be set to the new adapter vertex (step 4106), and the source port name will be set to Output.

The procedure for inserting a destination adapter vertex is symmetric, with "source" and "destination", and "Input" and "Output", respectively, being interchanged.

FIG. 49A and 49B illustrate the insertion of a communication adapter 4200 in place of link L between port P1 of vertex V1 and port P2 of vertex V2. In FIG. 42B, link L has been replaced by a source adapter 4200 including a first link 4202, a vertex 904, and a second link 4206. If the adapter 4200 is a file adapter, the vertex 4204 is a file vertex for a file in directory D1 on host H1, which are the directory and host of vertex

V1. If the adapter 4200 is a communication adapter, the vertex 4204 is a process vertex running a copy program. The second link 4202 is the original link L with its source vertex set to the adapter vertex 4204 and its source port name set to Output. The source port name of the first link 4202 is the set to the original source port name, P1, of the original link L, and its destination vertex is the Input port of the newly added vertex 4204.

Were the new adapter 4200 a destination adapter, the vertex 4204 would use the host H2 and directory D2 of the destination vertex V2 rather than the host H1 and directory D1 of the source vertex V1.

*Example of Inputting and Executing a Graph*

We will now consider this invention as applied to the payroll program described in FIG. 34. First, prior to creating the application, the user will have provided the system with templates for all required programs, specifically for the sort program, the data-checker program, and the payroll program. The user then expresses the application as a graph, as described above. The resulting graph is shown in FIG. 43, having vertices 4300-4370 running on nodes 0, 1, and 2 as shown (the names of the ports that various links are attached to are omitted, but should be obvious from comparison with FIG. 34B).

The first step in processing this graph is to prepare the graph for execution. First, while executing step 3750, we consult the template for the Sort program 4310, and observe that it requires a file as input. Therefore, step 3750 inserts a destination file adapter 4302 (FIG. 44).

We next execute step 3752, and note that we have a link from the Update file 4300 on node 2 to the Temporary File 4302 on node 1. Accordingly, both source and destination communication adapters (Copy 4304, Copy 4306) will be inserted into this link (FIG. 45).

Next, in step 3754 we identify complete file vertices (Update 4300 and Old Master 4350). Once this is done, in step 3756 we look for process vertices which are unrunnable because they have incomplete files upstream. The Sort program 4310 is found to satisfy this criteria. In step 3758 we identify process vertices which are runnable because they have only runnable process vertices and/or complete files upstream. The

two Copy programs 4304, 4306 (i.e., the communication adapters inserted in step 3752) meet this criteria, and are marked as runnable. All remaining nodes are marked as unrunnable. Finally, communication methods are chosen for those links attached to runnable processes vertices. The File method is chosen for those links attached to files, and TCP/IP is chosen for the link between the two Copy programs 4304, 4306 (which are on different nodes.) This leaves us with the situation shown in FIG. 46 (vertices not marked as runnable/complete are unrunnable/incomplete; links not marked with a communication method are unbound.)

The graph is now executable. A link file will be created (not shown), then the indicated links will be launched, and the runnable process vertices will be launched. When all processes have exited, the system will "clean up" by unlaunching the indicated links and changing the states of process vertices from "runnable" to "done". This results in the situation shown in FIG. 47.

The system now determines that not all processes are done, so it begins a new round of execution. First, it prepares the graph, as before. It starts by noting that the Temp File 4302 is complete. Next, it determines that there are no unrunnable process vertices. Finally, it determines that, in fact, all process vertices are runnable. This results in the situation shown in FIG. 48.

We are now ready to select communication methods (step 3764). First, the links attached to Temp File 4302, Bad 4330, and Paychecks 4360 are set to File, because they connect files and runnable process vertices and do not cross a node boundary. Second, the remaining links attached to Check 4320 are set to NamedPipe, because Check 4320 requires named pipes, and these links do not cross node boundaries.

This leaves us with the connections between OldMaster 4350, NewMaster 4370, and Process 4340, which connect files with a process running on a different node. Both cases require a communication adapter, which are inserted in step 3766 as Copy 4342 and Copy 4344. Both adapters run on Node 1. Once communication methods are chosen (File for links connecting to files, and TCP/IP for links crossing node boundaries), we have the graph shown in FIG. 49.

Again, the graph is ready for execution. The system will create a link file, launch links and processes, wait, and clean up. At this point, all process vertices are "done", so execution of the graph terminates.

**Program Implementation**

The invention may be implemented in hardware or software, or a combination of both. However, preferably, the invention is implemented in computer programs executing on programmable computers each comprising a processor, a data storage system (including volatile and non-volatile memory and/or storage elements), at least one input device, and at least one output device. Program code is applied to input data to perform the functions described herein and generate output information. The output information is applied to one or more output devices, in known fashion.

Each program is preferably implemented in a high level procedural or object oriented programming language to communicate with a computer system. However, the programs can be implemented in assembly or machine language, if desired. In any case, the language may be a compiled or interpreted language.

Each such computer program is preferably stored on a storage media or device (e.g., ROM or magnetic diskette) readable by a general or special purpose programmable computer, for configuring and operating the computer when the storage media or device is read by the computer to perform the procedures described herein. The inventive system may also be considered to be implemented as a computer-readable storage medium, configured with a computer program, where the storage medium so configured causes a computer to operate in a specific and predefined manner to perform the functions described herein.

A number of embodiments of the present invention have been described. Nevertheless, it will be understood that various modifications may be made without departing from the spirit and scope of the invention. Accordingly, it is to be understood that the invention is not to be limited by the specific illustrated embodiment, but only by the scope of the appended claims.

The impact analysis approach described above can be implemented using a computing system executing suitable software. For example, the software can include procedures in one or more computer programs that execute on one or more programmed or programmable computing system (which can be of various architectures such as distributed, client/server, or grid) each including at least one processor, at least one data storage system (including volatile and/or non-volatile memory and/or storage elements), at least one user interface (for receiving input using at least one input device or port, and for providing output using at least one output device or port). The software can include one or more modules of a larger program, for example, that provides services related to the design, configuration, and execution of dataflow graphs. The modules of the program (e.g., elements of a dataflow graph) can be implemented as data structures or other organized data conforming to a data model stored in a data repository.

The software can be provided on a tangible, non-transitory medium, such as a CD-ROM or other computer-readable medium (e.g., readable by a general or special purpose computing system or device), or delivered (e.g., encoded in a propagated signal) over a communication medium of a network to a tangible, non-transitory medium of a computing system where it is executed. Some or all of the processing can be performed on a special purpose computer, or using special-purpose hardware, such as coprocessors or field-programmable gate arrays (FPGAs) or dedicated, application-specific integrated circuits (ASICs). The processing can be implemented in a distributed manner in which different parts of the computation specified by the software are performed by different computing elements. Each such computer program is preferably stored on or downloaded to a computer-readable storage medium (e.g., solid state memory or media, or magnetic or optical media) of a storage device accessible by a general or special purpose programmable computer, for configuring and operating the computer when the storage device medium is read by the computer to perform the processing described herein. The inventive system can also be considered to be implemented as a tangible, non-transitory medium, configured with a computer program, where the medium so configured causes a computer to operate in a specific and predefined manner to perform one or more of the processing steps described herein.

A number of embodiments of the invention have been described. Nevertheless, it is to be understood that the foregoing description is intended to illustrate and not to limit the scope of the invention, which is defined by the scope of the following claims. Accordingly, other embodiments are also within the scope of the following claims. For example, various modifications can be made without departing from the scope of the invention. Additionally, some of the steps described above can be order independent, and thus can be performed in an order different from that described.

What is claimed is:

1. A computer-implemented method, the method including:

receiving data lineage information for multiple components representing at least two logical datasets and a transformation, with the data lineage information identifying a first logical dataset from which the transformation is to receive data and a second logical data dataset to which transformed data is to be provided, the transformation represented by a component in the data lineage information including one or more rules to be applied to data from the first logical dataset, and with the data lineage information identifying paths and flow traces of data though the multiple components;

receiving data specifying one or more proposed changes to a field in the first logical dataset, a field in the second logical dataset, or the transformation;

analyzing the multiple components according to the data lineage information to identify each component affected by the one or more proposed changes;

for at least one identified component identified as affected by the one or more proposed changes,

generating an impact metric representing a number of times a field specified by the one or more proposed changes is referenced within the at least one identified component;

determining, based on the generated impact metric, an impact of implementing the one or more proposed changes to one or more of the multiple components; and

storing information about the impact metric.

2. The method of claim 1, wherein the impact metric provides a measure of a direct impact of implementing the one or more proposed changes to the one or more of the multiple components.

3. The method of claim 1, wherein the impact metric provides a measure of an indirect impact of implementing the one or more proposed changes to the one or more of the multiple components.

4. The method of claim 1, further comprising associating a cost to a proposed change based on one or more impact metrics.

5. The method of claim 1, further including:

for that at least one component identified that represents the transformation, generating annotated code that includes the impact metric;

compiling the annotated code into an executable program; and

at least partially based on execution of the executable program, determining the impact of the one or more proposed changes to one or more of the multiple components.

6. The method of claim 1, further including:

selecting from data storage stored items of annotated code for one or more of the multiple components in the data lineage information, which are affected by the one or more proposed changes;

compiling the annotated code of one or more of the stored items into a transform; and

processing the transform to determine a risk output of a corresponding impact metric.

7. A system comprising:

one or more computers and one or more storage devices storing instructions that are operable, when executed by the one or more computers, to cause the one or more computers to perform operations comprising:

receiving a data lineage information for multiple components representing at least two logical datasets and a transformation, with the data lineage information identifying a first logical dataset describing characteristics of a first physical dataset from which the transformation is to receive data and a second logical dataset describing characteristics of a second physical dataset to which transformed data is to be provided, the transformation represented by a component in the data lineage information including one or more rules to be

applied to data from the first logical dataset, and with the data lineage information identifying paths and flow traces of data though the multiple components;

receiving data specifying one or more proposed changes to a field in the first logical dataset, a field in the second logical dataset, or the transformation;

analyzing the multiple components according to the data lineage information to identify each component affected by the one or more proposed changes;

for at least one component identified as affected by the one or more proposed changes,

generating an impact metric representing a number of times a field specified by the one or more proposed changes is referenced within the at least one identified component;

determining, based on the generated impact metric, an impact of implementing the one or more proposed changes to one or more of the multiple components; and

storing information about the impact metric.

8. The system of claim 7, wherein the impact metric provides a measure of a direct impact of implementing the one or more proposed changes to the one or more of the multiple components.

9. The system of claim 7, wherein the impact metric provides a measure of an indirect impact of implementing the one or more proposed changes to the one or more of the multiple components.

10. The system of claim 7, wherein a proposed change is one of the group consisting of a change in format of a field in a dataset or a change in a transformation.

11. The system of claim 7, wherein the operations further comprise associating a cost to a proposed change based on one or more impact metrics.

12. A system comprising:

one or more processor devices; and

memory operatively coupled to the one or more processor devices, storing a computer program that configures the system to:

receive data lineage information for multiple components representing at least first and second logical datasets and a transformation, with the data lineage information identifying a first physical dataset from which the transformation is to receive data and a second physical data dataset to which transformed data is to be provided, the transformation represented by a component in the data lineage information including one or more rules to be applied to data from the first logical dataset, and with the data lineage information identifying paths and flow traces of data though the multiple components;

receive data specifying one or more proposed changes to a field in the first logical dataset, the second logical dataset, or the transformation;

analyze the multiple components according to the data lineage information to identify each component affected by the one or more proposed changes;

for at least one identified component identified as affected by the one or more proposed changes,

generate an impact metric representing a number of times a field specified by the one or more proposed changes is referenced within the at least one identified component;

determine, based on the generated impact metric, an impact of implementing one or more of the multiple components; and

store information about the impact metric.

13. A computer storage medium encoded with computer program instructions that when executed by one or more computers cause the one or more computers to perform operations comprising:

receiving data lineage information for multiple components representing at least two logical datasets and a transformation, with the data lineage information identifying a first logical dataset describing characteristics of a first physical dataset from which the

transformation is to receive data and a second logical dataset describing characteristics of a second physical dataset to which transformed data is to be provided, the transformation represented by a component in the data lineage information including one or more rules to be applied to data from the first logical dataset, and with the data lineage information identifying paths and flow traces of data though the multiple components;

receiving data specifying one or more proposed changes to a field in the first logical dataset, a field in the second logical dataset, or the transformation;

analyzing the multiple components according to the data lineage information to identify each component affected by the one or more proposed changes;

for at least one component identified that represents a transformation,

for at least one identified component identified as affected by the one or more proposed changes,

generating an impact metric representing a number of times a field specified by the one or more proposed changes is referenced within the at least one identified component;

determining, based on the generated impact metric, an impact of implementing the one or more proposed changes to one or more of the multiple components; and

storing information about the impact metric.

14. The medium of claim 13, where the impact metric provides a measure of a direct impact of implementing the one or more proposed changes to the one or more of the multiple components.

15. The medium of claim 13, where the impact metric provides a measure an indirect impact of implementing the one or more proposed changes to the one or more of the multiple components.

16. The medium of claim 13, wherein a proposed change is one of the group consisting of a change in format of a field in a dataset or a change in a transformation.

17. The medium of claim 13, further comprising associating a cost to a proposed change based on one or more impact metrics.