

DISTRIBUTED SYSTEM GENERATING RULE COMPILER ENGINE APPARATUSES, METHODS, SYSTEMS AND MEDIA

[0001] This disclosure describes DISTRIBUTED SYSTEM GENERATING RULE COMPILER ENGINE APPARATUSES, METHODS, SYSTEMS AND MEDIA (hereinafter “DSGRCE”). A portion of the disclosure of this patent document contains material which is subject to copyright and/or mask work protection. The copyright and/or mask work owners have no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserve all copyright and mask work rights whatsoever.

CROSS-REFERENCE TO RELATED APPLICATIONS

[0002] Applicant hereby claims priority under 35 U.S.C. § 119 to United States provisional patent application no. 62/818,318, filed March 14, 2019, entitled “A DECLARATIVE SYNTAX AND RELATED METHODS, INTERFACE, ALGORITHMS AND PROCEDURES FOR BUILDING AND RUNNING SYSTEMS OF COMPLEX ANALYTICS”, docket no. LIY01.002.

[0003] Applicant hereby claims priority under 35 U.S.C. § 119 to United States provisional patent application no. 62/892,085, filed August 27, 2019, entitled “DECLARATIVE SYNTAX AND RELATED METHODS, INTERFACE,

1 ALGORITHMS AND PROCEDURES FOR BUILDING AND RUNNING
2 SYSTEMS OF COMPLEX ANALYTICS”, docket no. LIY01.004.

3 **[0004]** Applicant hereby claims priority under 35 U.S.C. § 119 to United States
4 provisional patent application no. 62/894,001, filed August 30, 2019, entitled
5 “DECLARATIVE SYNTAX AND RELATED METHODS, INTERLACE,
6 ALGORITHMS AND PROCEDURES FOR BUILDING AND RUNNING
7 SYSTEMS OF COMPLEX ANALYTICS”, docket no. 3200-101PV3.

8 **[0005]** The entire contents of the aforementioned applications are herein
9 expressly incorporated by reference in their entirety.

10 FIELD

11 **[0006]** The present disclosure is directed generally to compilers. Compilers are
12 an essential computer technology that allows computer-implemented generation
13 of processor-executable instructions from (e.g., human-readable) source code. The
14 DSGRCE implements improvements to the art.

15 BACKGROUND

16 **[0007]** A compiler translates computer code written in one programming
17 language into a different language.

18 BRIEF DESCRIPTION OF THE FIGURES

19 **[0008]** The accompanying figures and/or appendices illustrate various
20 exemplary embodiments in accordance with the present disclosure.

1 **[0009]** FIGUREs 1A-C show an exemplary architecture in one embodiment of
2 the DSGRCE.

3 **[0010]** FIGUREs 2A-B show a data flow diagram in one embodiment of the
4 DSGRCE.

5 **[0011]** FIGURE 3 shows a logic flow diagram illustrating a rule preprocessing
6 (RP) component in one embodiment of the DSGRCE.

7 **[0012]** FIGURE 4 shows a logic flow diagram illustrating a logical dependency
8 graph generating (LDGG) component in one embodiment of the DSGRCE.

9 **[0013]** FIGURE 5 shows a logic flow diagram illustrating a worker subgraph
10 distribution (WSD) component in one embodiment of the DSGRCE.

11 **[0014]** FIGURE 6 shows a logic flow diagram illustrating a worker
12 instantiation (WI) component in one embodiment of the DSGRCE.

13 **[0015]** FIGURE 7 shows a logic flow diagram illustrating a graph execution
14 (GE) component in one embodiment of the DSGRCE.

15 **[0016]** FIGUREs 8A-C show a logic flow diagram illustrating a worker
16 computation (WC) component in one embodiment of the DSGRCE.

17 **[0017]** FIGURE 9 shows a screen shot diagram illustrating user interface
18 features in one embodiment of the DSGRCE.

19 **[0018]** FIGURE 10 shows a screen shot diagram illustrating user interface
20 features in one embodiment of the DSGRCE.

1 **[0019]** FIGURE 11 shows a screen shot diagram illustrating user interface
2 features in one embodiment of the DSGRCE.

3 **[0020]** FIGURE 12 shows a screen shot diagram illustrating user interface
4 features in one embodiment of the DSGRCE.

5 **[0021]** FIGURE 13 shows a screen shot diagram illustrating user interface
6 features in one embodiment of the DSGRCE.

7 **[0022]** FIGURE 14 shows a screen shot diagram illustrating user interface
8 features in one embodiment of the DSGRCE.

9 **[0023]** FIGURE 15 shows a screen shot diagram illustrating user interface
10 features in one embodiment of the DSGRCE.

11 **[0024]** FIGURE 16 shows a screen shot diagram illustrating user interface
12 features in one embodiment of the DSGRCE.

13 **[0025]** FIGURE 17 shows a screen shot diagram illustrating user interface
14 features in one embodiment of the DSGRCE.

15 **[0026]** FIGURE 18 shows a screen shot diagram illustrating user interface
16 features in one embodiment of the DSGRCE.

17 **[0027]** FIGURE 19 shows a screen shot diagram illustrating user interface
18 features in one embodiment of the DSGRCE.

19 **[0028]** FIGURE 20 shows a screen shot diagram illustrating user interface
20 features in one embodiment of the DSGRCE.

1 **[0029]** FIGURE 21 shows a block diagram illustrating an exemplary DSGRCE
2 coordinator in one embodiment of the DSGRCE.

3 DETAILED DESCRIPTION

4 INTRODUCTION

5 **[0030]** The DSGRCE introduces a rule compiler that utilizes rule declarations
6 (rules) written in a domain specific language (DSL) and a library of atomic (e.g.,
7 with respect to the DSL) components (atoms) written in a hosting language to
8 create a fully functional and performing distributed system. For example, the
9 resulting distributed system may be utilized for solving numerical and analytical
10 problems. In some implementations, the DSL may depend on the hosting language
11 for basic language constructs, such as data types, common functions, and
12 operators. In some embodiments, rules may be utilized for implementing high
13 level business logic (e.g., ad hoc, fast changing and non-reusable), while atoms
14 may be utilized for implementing low level atomic numerical algorithms (e.g.,
15 stable and reusable).

16 **[0031]** In various embodiments, some of the benefits provided by the DSGRCE
17 may include:

- 18 1. The DSGRCE-generated system comprises virtual components. The virtual
19 components are not tied to any particular hardware or software, they
20 require minimal manual effort to create, configure, test and tune compared
21 to physical system components (e.g., components that are tied to specific

1 hardware, software and supporting personnel). Using such auto-generated
2 virtual components makes it easy and efficient to develop and test system
3 updates. Developers can focus their efforts on describing analytical logic
4 using rules and atoms, then use the rule compiler to re-generate the entire
5 distributed system and immediately see the results of the change.

6 2. The rule compiler is agnostic with respect to computing hardware and
7 environment, and supports parallel and distributed computing. The
8 DSGRCE-generated system may run on either a single computer, GPU, etc.,
9 or in parallel in a cloud computing environment for performance and
10 scalability.

11 3. The syntax of rules is rich enough to capture various system configurations
12 without separate configuration files for the system, which greatly simplifies
13 the release and deployment process, and facilitates speedy and reliable
14 updates to the entire system.

15 4. Arbitrary number of development (dev) and testing environments identical
16 to the production environment may be generated by the rule compiler
17 because system components are virtual.

18 5. As the set of rules and atoms defines current system capabilities, new
19 capabilities may be added by adding new rules and atoms. The
20 interdependencies of different applications and processes are automatically
21 taken care of by the rule compiler, simplifying project planning and
22 execution.

- 1 6. Easy integration and validation of in-house and 3rd party numerical
2 libraries, as they can be wrapped up as atoms using the programming
3 interface, and referred to by rules. As the intermediate computation results
4 of each atom are available for the developers or reviewers to inspect, the
5 validation process becomes easier.
- 6 7. The name space support of rules facilitates a multiteam development
7 environment. Different teams, departments or even vendors may develop,
8 test and release their rules independently in different name spaces, which
9 may then be seamlessly integrated by the rule compiler to form a complete
10 end-to-end system.
- 11 8. The ability to update the underlying rules independently from atoms, and
12 on a running system greatly simplifies the development, testing, release
13 and deployment process. Updates may also be made in a new rule
14 namespace, thus not changing the system behavior for existing users of the
15 old namespace.
- 16 9. Single version of truth and guaranteed repeatability – the behavior of the
17 DSGRCE-generated system is fully captured by the set of rules and the
18 underlying library of the atoms, from which the system behavior may be
19 exactly replicated. There is no need for separate configuration files to
20 change the system's behavior.
- 21 10. Intermediate computational results produced by any atom are readily
22 accessible in the resulting distributed system, giving users great

1 transparency to test, debug, audit and certify the correctness of complex
2 analytics.

- 3 11. The rule compiler may be configured to support full end-to-end algorithmic
4 differentiation (AD) from a tensor output. The distributed system generated
5 by the rule compiler may perform not only the primary calculation to
6 produce user specified output, but also the AD calculation, allowing fast
7 computation of sensitivities to any inputs.

8 **[0032]** Having a clear separation between rules and atoms allows for quick and
9 reliable update and release schedule in response to frequent business logic
10 changes. The conventional approach to implementing fast changing business logic
11 is to either implement such business logic in core quantitative libraries or as
12 configuration files that are read by applications at run time. Either choice has
13 significant drawbacks: Implementing high level business logic in a core
14 quantitative library is reliable but slow and inflexible, as every time when
15 business logic changes, a new version of the core quantitative library has to be
16 released by going through the full cycle of development, test, approval and
17 deployment; it is not an efficient use of valuable research and development
18 resources. Implementing high level business logic in configuration files allows
19 much faster release cycles, as many changes can be implemented by changing the
20 configuration files only. However, this approach incurs the additional cost and
21 complexity of having to manage the configuration files in addition to the analytics
22 libraries. In practice, the configuration files of complex analytics systems can often

1 grow into enormous complexity (e.g., different system components could require
2 different configuration formats and syntax; different versions of a numerical
3 library may require different configurations; the development and production
4 environments often also require different configurations) and require dedicated
5 database and software tools to manage. Such complexity in configuration is a
6 constant source of error in running and supporting complex analytics systems.
7 Further, fragmentation of the business logic across both the numerical library and
8 the collection of configuration files causes more difficulties in testing, debugging,
9 auditing and validations, as a developer or model reviewer has to go through not
10 only the source code of the numerical library, but also the myriads of
11 configurations to figure out the business logic in effect.

12 DETAILED DESCRIPTION OF THE DSGRCE

13 **[0033]** FIGURE 1A shows an exemplary architecture in one embodiment of the
14 DSGRCE. In Figure 1A, an embodiment of how a rule compiler 101 may be utilized
15 to facilitate creating and executing a distributed system is illustrated. The rule
16 compiler may utilize atoms 105 and rules 110 to create the distributed system. In
17 one embodiment, the DSGRCE may transforms rule specifications into the
18 distributed system in four phases.

19 **[0034]** In Phase 0, the rules may be parsed and data structures that link rules
20 and atoms, allow rule lookup, provide support for polymorphism based on type
21 inheritance and value matches, and/or the like may be generated. For example,
22 Phase 0 may be performed at compile time for compiled host languages such as

1 C/C++ and Fortran, and then dynamically loaded at run time as a shared library.
2 In another example, Phase 0 may be performed at run time for interpreted or just-
3 in-time compiled host languages, such as Python, Julia, Java, and .NET.

4 **[0035]** Phases 1 to 3 may be performed by the DSGRCE at run time. In Phase
5 1, a user may specify a desired output, which may be represented as a set of output
6 rules with concrete parameterization 115. The DSGRCE may analyze the set of
7 output rules and use the data structures generated in Phase 0 to construct a global
8 logical dependency graph 120 of the entire calculation in order to compute the
9 desired output. In one implementation, the global logical dependency graph (e.g.,
10 a directed acyclic graph (DAG)) is constructed by recursively following and adding
11 rule precedences (e.g., data precedences of the primary output calculation) as new
12 nodes to the global logical dependency graph, starting from the set of output rules
13 requested by the user. At the end of Phase 1, the atom objects have not been
14 instantiated and the memory utilized for the calculation (e.g., tensor calculation)
15 has not been allocated. Thus, each node in the graph stores minimal information
16 and extremely large graphs, with hundreds of millions of nodes, can comfortably
17 fit into the memory of a modern PC.

18 **[0036]** In Phase 2, a manager process may divide the global logical dependency
19 graph into multiple sub-graphs according to the number of available worker
20 processes and their hardware resources. In one implementation, the manager
21 process and/or the worker processes may be executed in a cloud computing system
22 125. The manager process may instantiate the worker processes and/or may

1 facilitate establishing communication links (e.g., network communication links)
2 among worker processes. It is to be understood that while a distributed computing
3 topology with a single manager and multiple workers is utilized in this
4 embodiment, the DSGRCE may utilize a wide variety of topologies, such as
5 multiple layers of managers and/or workers, or a fully decentralized network of
6 instances, which may be either managers or workers, where a manager may be
7 selected by voting or polling of workers. At the end of Phase 2, the atom objects
8 have been created on individual workers.

9 **[0037]** In Phase 3, the manager process coordinates the execution of the graph
10 calculation, and obtains the final results. In one embodiment, the execution may
11 comprise two passes, a forward pass (or primary pass) to compute the value of the
12 requested output, and a backward pass to compute the first order derivatives
13 using adjoint algorithmic differentiation (AAD) if user also wants sensitivities. In
14 another embodiment, the execution may comprise a single forward pass to
15 compute the value of the requested output and forward algorithmic
16 differentiation. In another embodiment, the execution may comprise two passes,
17 a first forward pass to compute the value of the requested output, and a second
18 forward pass to compute forward algorithmic differentiation. In some
19 implementations, steps of execution don't have to be fully synchronized, for
20 example, the manager may message a worker to start the backward AAD
21 initialization, even when other workers are still running the forward calculation.
22 However, a worker may block when it can no longer making any progress on its

1 sub-graph, which may happen when it hasn't received the preceding inputs from
2 other workers. Both managers and workers may run multiple tasks concurrently,
3 so that they can send/receive messages while simultaneously running
4 calculations. The concurrency may be implemented using multi-process, multi-
5 threading, co-routines, or green threads. Once the computation is complete, the
6 manager may message the right worker process to retrieve the results for any
7 nodes. For example, the resulting distributed system may be utilized for solving
8 numerical and analytical problems, such as those commonly found in scientific
9 research, engineering, data science, statistical forecasting, simulation, time series
10 analysis, business and financial modeling, and/or the like.

11 **[0038]** FIGUREs 1B-C show an exemplary architecture in one embodiment of
12 the DSGRCE. In Figures 1B-C, an embodiment of how a set of rules may be
13 declared is illustrated. Declarations 130 may be used to describe a set of rules for
14 calculating a Fibonacci series using recursion and pattern matching by values. A
15 “@rules” indicator 135 signifies that a set of rules is being declared. A namespace
16 140 indicates that “series” is the namespace for this set of rules. In one
17 implementation, rules from different namespaces may be referenced using their
18 fully qualified names. For example, rules from other namespaces may refer to this
19 set of rules using “series.fib”. The “series” namespace includes three rules 142,
20 144, 146. In one implementation, a rule may be uniquely identified by the triplet
21 **[namespace, rulename, ruleparameters types/values]**, where rule parameter
22 types/values may be any data types/values defined in the hosting language of the

DSL. For example, with regard to rule 142, the rule name 150 is “fib” and the rule has one parameter 152 of type “Int”, which can match any integer value. In one implementation, each rule may specify an atom (e.g., a concrete **DerivedAtom** that implements a generic **Atom** interface described later) and a list of its constructor parameters. For example, with regard to rule 142, the atom name 154 is “Sum” and the atom does not take any parameters (e.g., square brackets may be omitted). In another example, with regard to rule 144, the atom name 156 is “Const” and the atom takes a number as a parameter (e.g., **Atom** parameters are listed in order in square brackets). Each rule may identify a list of its precedences, which are the rules whose outputs (e.g., output tensors) are used as inputs (e.g., input tensors) to the current rule’s forward pass computation function (e.g., **fwd0**). Some rules may not have any precedences (e.g., accessing data sources). For example, with regard to rule 142, the rule has two precedences 158 “fib(n-1)” and “fib(n-2)” that may recursively refer to rule 142 or to rule 144 or to rule 146 depending on the value of “n”.

[0039] In various implementations, rule declarations may support the following features:

1. Direct access to hosting language features: a rule may use hosting languages syntax and features in any of the parameter fields. For example, with regard to a rule such as:

```
pv(basket::Basket,
env::Env)=WeightedSum[basket.notionals]([pv(i, env) for
i=basket.insts]...)
```

1

2 “basket.insts” may be a syntax in the hosting language to access a data field
3 of an object. Similarly, function calls and/or the hosting language’s types
4 (e.g., including user-defined types) may be utilized in rule declarations
5 using the hosting language syntax.

6 **2.** Dynamic precedence: Rule precedence may be dynamic. For example, a
7 declaration may use a list comprehension syntax such as “[pv(i) for
8 i=basket.insts]...” to specify that the present value (pv) of a basket depends
9 on the pv of every individual trade in the basket. The number and identity
10 of precedence can depend on the rule parameters. This allows the creation
11 of different logical graph topologies from the same set of rules when given
12 different rule parameters. Besides the list comprehension, the full syntax
13 features of the hosting language may be used for greater flexibility (e.g., the
14 “...” syntax may be used with regular function calls in the hosting language
15 to return a list of precedences). In one implementation, dynamic precedents
16 of rules may be implemented by directly calling such dynamic precedence
17 code (e.g., in the hosting language) at run time (e.g., when the rule engine
18 is expanding the logical dependency graph).

19 **3.** Overloading and run time polymorphism by type and/or value: rules “fib”
20 142, 144, 146 are declared multiple times for different parameter types and
21 values. At run time, the rule with the closest match to all the input
22 parameters’ types and values would take effect. In one implementation, the

1 closeness of parameter types may be measured by the distance of their type
2 inheritance. The rule overloading and pattern matching allow complicated
3 business logic to be expressed succinctly.

4 **4.** Recursion: a rule can reference to itself as precedence, as shown with regard
5 to rule 142.

6 **5.** Clone of namespace: the entire rules in a namespace may be cloned into
7 another namespace. In one implementation, this results in two identical
8 sets of rules in different namespaces. For example, with regard to
9 declarations 160, any reference to namespace “series:dev” is created as a
10 clone of namespace “series” using the “@clone” syntax 162. Additional rules
11 166 (e.g., that are not present in the old namespace) may be added to a
12 cloned namespace. Such newly added rules may override existing rules
13 cloned from the old namespace with the same rule name and parameters
14 signature. In one implementation, namespace cloning may be implemented
15 by adding a generic (e.g., matching for any **rulename**) entry in the rule
16 lookup table. For example, when cloning namespace A to namespace B, a
17 generic rule entry for namespace B is inserted into the rule lookup table
18 data structure (e.g., this rule entry matches any reference to namespace B
19 after the cloning). The rule compiler automatically generates code such that
20 any reference to this entry B redirects to a rule lookup using namespace A,
21 with rule names and other run time rule parameters unchanged.

1 **6.** Redirection of namespace: any reference to a namespace may be redirected
2 to use another namespace. For example, with regard to declarations 160,
3 any reference to namespace “series” may be redirected to use a rule of the
4 same name under namespace “series:dev”. Such redirection may use the
5 syntax of the hosting language, and may be changed multiple times at run
6 time. For example, in C++ syntax, the redirect call may look like:

```
7       namespaceRedirect(config, "series", "series:dev")
```

8
9 where config is a meta parameter variable that is used at run time by the
10 rule engine to create and execute the graph. For example, the clone and
11 redirect features may allow a developer to create multiple meta parameters
12 to compare the results between different redirections of namespaces side by
13 side (e.g., to see how a change to an existing system affects results). In one
14 implementation, namespace redirection may be implemented as one more
15 layer of redirection (e.g., using an additional namespace lookup table, or the
16 rule lookup table), which translates a namespace reference to the real
17 namespace used for rule lookup. In the example discussed with regard to
18 namespace cloning, after namespace A is cloned to namespace B, a user
19 may add or override additional rules in namespace B. Afterwards, the user
20 may redirect namespace A to B to force any reference to namespace A to
21 use namespace B's overriding or additional rules. After such redirection
22 instructions, a reference to a rule in namespace A is redirected to
23 corresponding rule in namespace B via the namespace lookup table (e.g.,

1 implemented as a map of identification tuple (e.g., **[namespace]** tuple) to
2 identification tuple (e.g., **[namespace]** tuple)), and the rule lookup is
3 performed using namespace B. If the rule matches an overridden or new
4 rule in namespace B, then the rule is chosen, otherwise, the generic rule
5 entry for cloning namespace B matches and it falls back to the original rule
6 declared in namespace A.

7 **7. Package:** a set of rules may be packaged together into a **PackagedAtom**,
8 which supports the **Atom** API and behaves like a regular atom (e.g., it may
9 be referred by other rules). This allows hiding and encapsulation of low-
10 level logic (e.g., complex logic may be constructed by recursively
11 interleaving rules and atoms). For example, declarations 170 may be used
12 to describe a set of rules for calculating a Fibonacci series, but using a
13 packaged atom that allows Tensors instead of scalars to be used as initial
14 values.

15 **[0040]** In some implementations, the rule DSL may exclude a set of
16 programming language features such as functions, types, variables, loops,
17 branches, exceptions, and/or the like (e.g., which may exist in the hosting
18 language). The minimal set of syntax features may facilitate analysis of the overall
19 logic defined by the rules and creation of an optimized distributed system that
20 implements the correct logic by the DSGRCE. Further, the minimal set of syntax
21 features makes the rules easy for non-programmers to understand and author.

[0041] In one embodiment, an atom may be a low level atomic numerical algorithm implemented in the hosting language. In one implementation, an atom represents a generic mathematical function of:

$$(\overrightarrow{y_1}, \overrightarrow{y_2}, \dots, \overrightarrow{y_m}) = f(\overrightarrow{x_1}, \overrightarrow{x_2}, \dots, \overrightarrow{x_n})$$

whose inputs are n tensors of $\overrightarrow{x_i}$, and outputs are m tensors of $\overrightarrow{y_j}$. A tensor is a multi-dimensional array of numbers, and its elements are indexed by a vector of integers. The dimensionality of a tensor is commonly referred as its rank. A tensor of rank 0, 1, 2 is scalar, vector and matrix respectively. Each input and output tensor may have different ranks and dimensions.

[0042] In one implementation, an atom is atomic with respect to the DSL (e.g., with the exception of the aforementioned packaged atoms, which behave like regular atoms but have inner structure that a developer may inspect), it carries out a potentially sophisticated numerical calculation in the hosting language, such as root search, optimization or partial differential equation (PDE) solver, and/or the like.

[0043] In one implementation, an atom encapsulates a generic interface **Atom** in the hosting language. Concrete derived types of **DerivedAtom** may inherit from the **Atom** interface, and may implement the following operations or methods from the generic interface **Atom**:

- **DerivedAtom(xdim::Collection, ps::Collection)**: the creation function, or commonly referred to as constructor or factory function, of an instance of the

1 type **DerivedAtom**. The **xdim** and **ps** are the parameters to the **DerivedAtom**
 2 constructor, the **xdim** is the input tensor dimensions, whereas the parameter
 3 **ps** is a collection of parameters for creating a **DerivedAtom** instance, which
 4 may include any data type supported by the hosting programming language.
 5 Two atom objects are logically identical and refer to the same mathematical
 6 function if they have identical **DerivedAtom** and parameters **xdim** and **ps**.

7 • **dim(a::Atom)**: returns the rank and dimension of the input and output tensors
 8 of an atom **a**, which was created by the **DerivedAtom** constructor.

9 • **fwd(a::Atom, xs::Vector{Tensor})**: the forward calculation operation of the
 10 atom, which corresponds to the mathematical function of
 11 $(\overrightarrow{y_1}, \overrightarrow{y_2}, \dots, \overrightarrow{y_m}) = a(\overrightarrow{x_1}, \overrightarrow{x_2}, \dots, \overrightarrow{x_n})$, where the object **a** is an instance of Atom created
 12 by a constructor.

13 • **back(a::Atom, ys::Vector{Tensor}, xs::Vector{Tensor}, jacys::Vector{Tensor},**
 14 **jacxs::Vector{Tensor})**: the backward calculation operation of the atom, which
 15 is utilized for backward AAD. Its inputs are: **xs** = $(\overrightarrow{x_1}, \overrightarrow{x_2}, \dots, \overrightarrow{x_n})$, **ys** =
 16 $(\overrightarrow{y_1}, \overrightarrow{y_2}, \dots, \overrightarrow{y_n})$, **jacys** = $(\frac{\partial \vec{u}}{\partial y_1}, \frac{\partial \vec{u}}{\partial y_2}, \dots, \frac{\partial \vec{u}}{\partial y_n})$, and its output is **jacxs** = $(\frac{\partial \vec{u}}{\partial x_1}, \frac{\partial \vec{u}}{\partial x_2}, \dots, \frac{\partial \vec{u}}{\partial x_n})$,

17 where **xs** and **ys** are already computed from the **fwd()** method. The tensor $\vec{u}$ is
 18 specified by the user and may be any (e.g., intermediate) results that the user
 19 wishes to analyze for sensitivities, and which may depend on both **ys** and **xs**;

20 $\frac{\partial \vec{a}}{\partial \vec{b}}$ represents the Jacobian tensor of $\vec{a}$ to $\vec{b}$, whose rank is the sum of the ranks
 21 of $\vec{a}$, $\vec{b}$. In some embodiments, a default implementation of **back()** for the base

1 **Atom** type may be provided using numerical finite difference as a fall back if
2 the developer did not provide a fast implementation of analytical **back()**
3 function. In some embodiments, the Atom interface may have a separate
4 **fwdad()** method that has identical function signature as the **back()** method,
5 which is used for automatic forward AD implementation. The main difference
6 in **fwdad()** method is that the **jacxs** is input and **jacys** is output, which is the
7 opposite of the **back()** function.

8 **[0044]** In one implementation, a **DerivedAtom** may be implemented as a user
9 defined data type in object-oriented programming languages, where the **Atom**
10 interfaces would be virtual functions or methods of the **DerivedAtom**. The function
11 names, such as **fwd**, **back**, and **fwdad**, are insignificant and may differ in various
12 implementations.

13 **[0045]** In one embodiment, a **PackagedAtom** may be utilized to encapsulate
14 complicated analytical logic into a single atom, leading to significant
15 improvements in system's consistency, transparency and logical clarity. For
16 example, while low level numerical algorithms, such as weighted average and
17 regression, may be implemented using an atom, more sophisticated numerical
18 algorithms that are typically used as a single atomic algorithm, such as the cubic
19 spline interpolation, may be implemented using another atom or multiple atoms
20 (e.g., a cubic spline interpolation atom may be implemented by declaring a few
21 rules that include a Thomas Algorithm atom). While this approach is simple,
22 reusable and consistent, the inner logic (e.g., of the cubic spline interpolation

1 algorithm) is fully exposed to users (e.g., who may not care or need to know about
2 the inner logic of the cubic spline interpolation algorithm provided for their use).
3 Fully expanding and exposing the inner logic complicates the resulting logical
4 dependency graph, and may make it more difficult and tedious for a typical user
5 to visualize, navigate and understand the higher level business and analytical
6 logic the user is working on.

7 **[0046]** In one implementation, the DSGRCE supports a special syntax (e.g.,
8 @package) to allow a developer to declare a packaged atom (e.g., see declarations
9 170) using rules. The DSGRCE may automatically create a packaged atom from
10 such declarations using a **PackagedAtom** type that may inherit from the **Atom**
11 interface. A **PackagedAtom** may be used and referenced by rules like any other
12 **Atom**, and may appear to a typical user as an “atomic” unit. However, a
13 **PackagedAtom** may have its own private internal logical dependency graph, which
14 is automatically created at run time by the DSGRCE. For example, the cubic
15 spline interpolation **PackagedAtom** may have an internal logical dependency
16 graph that references a Thomas Algorithm **DerivedAtom**. In one implementation,
17 the DSGRCE may provide GUI support for users to drill down to the inner graph
18 of a **PackagedAtom**, giving full access to those users who wish to inspect the data
19 and/or logical lineage inside a packaged atom. A **PackagedAtom** may implement
20 operations or methods from the generic interface **Atom** as follows:

- 21 • **PackagedAtom**: the creation function, or commonly referred to as constructor
22 or factory function, of an instance of the type **PackagedAtom**. The declaration

1 of a packaged atom specifies a list of input rules (e.g., the parameter list at
2 170). The DSGRCE retrieves these input rules, creates an in-memory logical
3 dependency graph, and adds nodes corresponding to each input rule, without
4 adding their precedents that are not part of the inputs. If an input node
5 depends on another input node, an edge is added between them to capture the
6 dependency. The result is an inner logical dependency graph that contains the
7 input nodes and edges between them. The declaration of the packaged atom
8 also specifies an output rule (e.g., the rule in the last return statement at 170).
9 The output node and its precedents are recursively added to the inner logical
10 dependency graph. The recursion stops at any of the input nodes, so that the
11 input nodes' precedents are not added to the logical graph. Thus, the DSGRCE
12 constructs a logical dependency graph that starts from input nodes, and
13 produce the output node. If the output node has any (e.g., recursive) precedent
14 that is not a dependent of any of the input node, an error may be thrown as the
15 list of inputs is not sufficient to uniquely determine the result of the output
16 nodes. The user may fix the error by adding additional inputs to the packaged
17 atom declaration. Once the logical dependency graph is created, the
18 corresponding atoms may be instantiated in memory by traversing the graph,
19 and memory utilized for subsequent calculations may be pre-allocated.

20 • **dim**: the input and output atoms' **dim()** methods may be called to determine
21 the proper input and output size of the entire calculation.

- 1 • **fwd**: the **xs** value in the argument of the **fwd()** call may be copied (or linked) to
2 the pre-allocated memory of the inner graph, and the inner graph may be
3 traversed accordingly following its dependency. For each atom in the inner
4 graph, the atom's **fwd()** method may be called, until the final output is
5 computed. The result of the final output may be copied to the **ys** argument in
6 the **fwd()** call.
- 7 • **back**: the values in the **jacys**, **ys**, and **xs** arguments of **back()** may be copied (or
8 linked) to the pre-allocated memory of the inner graph, and **jacxs** matrices may
9 be initialized to zeroes. The inner graph may be traversed in the opposite order
10 of the **fwd()** calculation, and each atom's **back()** method may be called in turn
11 to propagate the Jacobian matrix backwards, until the input nodes have been
12 processed. Then the resulting Jacobians to the input nodes may be added to
13 the corresponding **jacxs** of the arguments to the **back()** of the **PackagedAtom**.
14 Alternatively, the **jacxs** values in the **back()** argument may be copied to the pre-
15 allocated memory of the Jacobians to the inputs of the inner graph at the
16 beginning of **back()**, and then the resulting Jacobian to the inputs of the inner
17 graph (after the backward run) may be copied to the **jacxs** of the
18 **PackagedAtom**.

19 In one implementation, the entire inner graph of a packaged atom may be in the
20 same memory space as the calling process (e.g., a worker process). In another
21 implementation, the inner graph of a packaged atom may be distributed across
22 multiple processes or computers connected by a network in a similar manner as

described with regard to distributing a logical dependency graph. The inner graph of a packaged atom may reference any atom, including another packaged atom. Therefore, a user may create a hierarchy of packaged atoms with arbitrary depth to express and encapsulate very complicated business and/or analytical logic, while simplifying the logic and data lineage view at each layer of the hierarchy. In one implementation, the DSGRCE may provide GUI support for users to navigate through the layers of the packaged atoms to access and/or inspect the data and/or logic of any layer.

[0047] Even though a tensor is the most generic input and output type of an atom, in some implementations, the programming interface and implementation may be simplified by using low ranking tensors for inputs and/or outputs. For example, an implementation could choose to restrict **xs** and **ys** inputs/outputs to vectors (rank 1 tensor). In such an implementation, higher ranked tensor inputs and/or outputs may be encoded as a collection of vectors.

[0048] FIGURES 2A-B show a data flow diagram in one embodiment of the DSGRCE. Figures 2A-B provide an example of how data may flow to, through, and/or from the DSGRCE. In Figures 2A-B, a rule preprocessing (RP) component 221 may parse a set of rules to generate data structures that link rules and atoms, allow rule lookup, provide support for value and/or type polymorphism, and/or the like. See Figure 3 for additional details regarding the RP component.

[0049] A user 202 may send a distributed system execution request input to a DSGRCE server 206 to facilitate execution of a specified calculation using the set

1 of rules. In one embodiment, the user may use a client (e.g., a desktop, a laptop, a
2 tablet, a smartphone, a smartwatch) to access a DSGRCE website, application
3 (e.g., a desktop application, a mobile app), and/or the like to facilitate generating
4 the distributed system execution request input. In one implementation, the
5 distributed system execution request input may include data such as a request
6 identifier, user credentials, a calculation specification, and/or the like. For
7 example, the distributed system execution request input may be sent via a
8 network in XML format substantially in the following form:

```
9      <?XML version = "1.0" encoding = "UTF-8"?>
10      <distributed_system_execution_request_input>
11          <request_identifier>ID_request_1</request_identifier>
12          <user_identifier>ID_user_1</user_identifier>
13          <credential_token>user's credential token for authentication</credential_token>
14          <calculation_specification>series.fib(10)</calculation_specification>
15      </distributed_system_execution_request_input>
```

16
17 **[0050]** The DSGRCE server 206 may send a distributed system execution
18 request 229 to a manager process 210 to facilitate generating a distributed system
19 that executes the specified calculation. In one embodiment, the manager process
20 may be a separate process from the DSGRCE server (e.g., running on different
21 hardware). In another embodiment, the manager process may be a component of
22 the DSGRCE server (e.g., running on the same hardware in the same process and
23 memory space as the DSGRCE server). In one implementation, the distributed
24 system execution request may include data such as a request identifier, a

1 calculation specification, an execution complexity gauge, available workers
2 specification, communications overhead settings, and/or the like. For example, the
3 distributed system execution request may be sent via a network in XML format
4 substantially in the following form:

```
5      <?XML version = "1.0" encoding = "UTF-8"?>
6      <distributed_system_execution_request>
7          <request_identifier>ID_request_2</request_identifier>
8          <calculation_specification>series.fib(10)</calculation_specification>
9          <execution_complexity_gauge>
10             NETWORK_DATA_TRANSFER_SIZE
11          </execution_complexity_gauge>
12          <available_workers_specification>
13             <type>FIXED</type>
14             <number_of_available_workers>6</number_of_available_workers>
15          </available_workers_specification>
16          <communications_overhead>DUPLICATE_HOT_NODES</communications_overhead>
17      </distributed_system_execution_request>
```

19 **[0051]** A logical dependency graph generating (LDGG) component 233 may
20 utilize data provided in the distributed system execution request to facilitate
21 constructing a global logical dependency graph (LDG) of the specified calculation.
22 See Figure 4 for additional details regarding the LDGG component.

23 **[0052]** A worker subgraph distribution (WSD) component 237 may divide the
24 global logical dependency graph into multiple sub-graphs (e.g., according to the
25 number of available worker processes and their hardware resources) and/or may

1 instantiate worker processes. See Figure 5 for additional details regarding the
2 WSD component.

3 **[0053]** The manager process 210 may send one or more instantiation requests
4 241 to one or more worker processes 214 to instantiate the worker processes and/or
5 to provide each worker process with its respective sub-graph of the global logical
6 dependency graph. In some embodiments, the worker processes may execute in a
7 cloud computing environment (e.g., the manager process and/or the DSGRCE
8 server may also execute in the cloud computing environment or may execute in a
9 separate environment). In one implementation, an instantiation request may
10 include data such as a request identifier, a worker process identifier, an LDG
11 subgraph specification, a communication mode, workers network configuration,
12 shared cache network configuration, subgraph nodes remote precedences and/or
13 dependencies specification, and/or the like. For example, the instantiation request
14 may be sent via a network in XML format substantially in the following form:

```
15 <?XML version = "1.0" encoding = "UTF-8"?>
16 <instantiation_request>
17     <request_identifier>ID_request_3</request_identifier>
18     <worker_process_identifier>ID_worker_1</worker_process_identifier>
19     <LDG_subgraph>LDG subgraph specification (e.g., in DAG format)</LDG_subgraph>
20     <communication_mode>WORKER_TO_WORKER</communication_mode>
21     <workers_network_configuration>
22         <worker>
23             <worker_process_identifier>ID_worker_1</worker_process_identifier>
24             <worker_IP_address>10.0.0.155:6777</worker_IP_address>
25         </worker>
```

```
1          <worker>
2              <worker_process_identfier>ID_worker_2</worker_process_identfier>
3              <worker_IP_address>10.0.0.57:6777</worker_IP_address>
4          </worker>
5          <worker>
6              <worker_process_identfier>ID_worker_3</worker_process_identfier>
7              <worker_IP_address>10.0.0.175:6777</worker_IP_address>
8          </worker>
9          ...
10     </workers_network_configuration>
11     <subgraph_nodes_links>
12         <node>
13             <node_identfier>ID_node_11</node_identfier>
14             <precedences>
15                 ID_worker_2:ID_Node_22, ID_worker_3:ID_Node_35
16             </precedences>
17             <dependencies>ID_worker_6:ID_Node_62</dependencies>
18         </node>
19         <node>
20             <node_identfier>ID_node_12</node_identfier>
21             <precedences>NONE</precedences>
22             <dependencies>
23                 ID_worker_5:ID_Node_53, ID_worker_6:ID_Node_66
24             </dependencies>
25         </node>
26         <node>
27             <node_identfier>ID_node_13</node_identfier>
28             <precedences>NONE</precedences>
29             <dependencies>NONE</dependencies>
30         </node>
31         ...
32     </subgraph_nodes_links>
```

1 </instantiation_request>
2

3 It is to be understood that, in some implementations, a binary representation may
4 be used (e.g., instead of XML) when sending DSGRCE messages (e.g., inputs,
5 outputs, requests, responses) to reduce network data transfer.

6 **[0054]** A worker instantiation (WI) component 245 may utilize data provided
7 in an instantiation request to instantiate the respective worker process and/or to
8 establish communication links with other worker processes (e.g., that are remote
9 precedences and/or remote dependencies). See Figure 6 for additional details
10 regarding the WI component.

11 **[0055]** The one or more worker processes 214 may send one or more
12 instantiation responses 249 to the manager process 210 to inform the manager
13 process that a respective worker process instantiated successfully. In one
14 implementation, an instantiation response may include data such as a response
15 identifier, a status, and/or the like. For example, the instantiation response may
16 be sent via a network in XML format substantially in the following form:

17 <?XML version = "1.0" encoding = "UTF-8"?>
18 <instantiation_response>
19 <response_identifier>ID_response_3</response_identifier>
20 <status>OK</status>
21 </instantiation_response>
22

1 **[0056]** A graph execution (GE) component 253 may coordinate the execution of
2 the specified calculation and/or provide the final results. See Figure 7 for
3 additional details regarding the GE component.

4 **[0057]** The manager process 210 may send one or more computation requests
5 257 to the one or more worker processes 214 to instruct a set of worker processes
6 to perform a computation. In one implementation, a computation request may
7 include data such as a request identifier, a worker process identifier, a request
8 type, subgraph nodes to run, and/or the like. For example, the computation
9 request may be sent via a network in XML format substantially in the following
10 form:

```
11       <?XML version = "1.0" encoding = "UTF-8"?>
12       <computation_request>
13           <request_identifier>ID_request_4</request_identifier>
14           <worker_process_identifier>ID_worker_1</worker_process_identifier>
15           <request_type>CALCULATE_FORWARD</request_type>
16           <subgraph_nodes_to_run> ID_node_12, ID_node_13, ...</subgraph_nodes_to_run>
17       </computation_request>
```

19 **[0058]** A worker computation (WC) component 261 may utilize data provided
20 in a computation request to facilitate execution of the requested computation by
21 the respective worker process. See Figures 8A-C for additional details regarding
22 the WC component.

23 **[0059]** The worker processes 214 may send a computation response 265 to the
24 manager process 210 with results of the requested computation (e.g., a

confirmation that initialization was completed successfully; the final results of the specified calculation). In one implementation, the computation response may include data such as a response identifier, a status, the requested computation results, and/or the like. For example, the computation response may be sent via a network in XML format substantially in the following form:

```
<?XML version = "1.0" encoding = "UTF-8"?>
<computation_response>
  <response_identifier>ID_response_4</response_identifier>
  <status>OK</status>
  <computation_results>[55; 30; 73]</computation_results>
</computation_response>
```

In an alternative implementation, the computation response may be used to provide the status, and separate computation result retrieval request/response messages may be utilized by the manager process and the worker processes to transfer computation results. Such separation between execution and data retrieval may allow the manager process to retrieve previously computed data without trigger a recalculation by sending a computation result retrieval request.

[0060] The manager process 210 may send a distributed system execution response 269 to the DSGRCE server 206 with results for the specified calculation.

In one implementation, the distributed system execution response may include data such as a response identifier, a status, the requested calculation results, and/or the like. For example, the distributed system execution response may be sent via a network in XML format substantially in the following form:

```
1      <?XML version = "1.0" encoding = "UTF-8"?>
2      <distributed_system_execution_response>
3          <response_identifier>ID_response_2</response_identifier>
4          <status>OK</status>
5          <calculation_results>[55; 30; 73]</calculation_results>
6          <sensitivities>sensitivities for the calculation</sensitivities>
7      </distributed_system_execution_response>
8
```

9 In an alternative implementation, the distributed system execution response may
10 be used to provide the status, and separate computation result retrieval
11 request/response messages may be utilized by the DSGRCE server and the
12 manager process to transfer computation results. Such separation between
13 execution and data retrieval may allow the DSGRCE server to retrieve previously
14 computed data without trigger a recalculation by sending a computation result
15 retrieval request.

16 **[0061]** The DSGRCE server 206 may send a distributed system execution
17 response output 273 to the user 202 with results for the specified calculation. In
18 one embodiment, the user's client may be instructed to display the requested
19 results. In one implementation, the distributed system execution response output
20 may include data such as a response identifier, a status, the requested calculation
21 results, and/or the like. For example, the distributed system execution response
22 output may be sent via a network in XML format substantially in the following
23 form:

```
24      <?XML version = "1.0" encoding = "UTF-8"?>
```

```
1      <distributed_system_execution_response_output>
2          <response_identifier>ID_response_1</response_identifier>
3          <status>OK</status>
4          <calculation_results>[55; 30; 73]</calculation_results>
5          <sensitivities>sensitivities for the calculation</sensitivities>
6      </distributed_system_execution_response_output>
```

7

8 In an alternative implementation, the distributed system execution response

9 output may be used to provide the status, and separate computation result

10 retrieval request input/response output messages may be utilized by the user and

11 the DSGRCE server to query computation results. Such separation between

12 execution and data retrieval may allow the user to retrieve previously computed

13 data without trigger a recalculation by sending a computation result retrieval

14 request input.

15 **[0062]** FIGURE 3 shows a logic flow diagram illustrating a rule preprocessing

16 (RP) component in one embodiment of the DSGRCE. In Figure 3, a preprocessing

17 request may be obtained at 301. For example, the preprocessing request may be

18 obtained as a result of compilation (e.g., requested by a developer user) of a set of

19 specified rules.

20 **[0063]** The set of specified rules may be determined at 305. In one

21 implementation, one or more specified files with rule declarations written in the

22 DSL may be parsed to determine the set of specified rules.

23 **[0064]** A determination may be made at 309 whether there remain rules to

24 process. In one implementation, each of the rules in the set of specified rules may

1 be processed. If there remain rules to process, the next rule may be selected for
2 processing at 313.

3 **[0065]** A rule data structure instance for the selected rule may be generated at
4 317. In one implementation, the rule data structure may be a **Rule** type that stores
5 information regarding the selected rule, such as namespace, name, rule
6 parameters, atom name, atom parameters, precedent rules, meta data (e.g.,
7 author, copyright, developer, supporter, documentation URL), and/or the like,
8 which may be generated based on parsing the selected rule's declaration. For
9 example, information in rule data structures (e.g., precedent rules) may be used
10 to construct a rule dependency graph (e.g., by recursively traversing through the
11 precedent rules).

12 **[0066]** The rule data structure instance for the selected rule may be added to a
13 rule lookup table at 321. For example, the rule lookup table may be utilized to
14 facilitate quick lookup of declared **Rule** instances. In one implementation, the rule
15 lookup table may be keyed by a rule's identification tuple (e.g., **[namespace,**
16 **rulename]** tuple) to a set (e.g., a list) of **Rule** instances. For example, a reference
17 (or pointer) to the selected rule's **Rule** instance may be added to the list of
18 references associated with the selected rule's **[namespace, rulename]** tuple. In
19 some alternative implementations, the rule lookup table may instead be keyed by
20 a rule's hash value calculated based on a rule's identification tuple.

21 **[0067]** A factory function for the selected rule may be generated at 325. In one
22 implementation, a factory function that takes the selected rule's parameters as

1 input and creates a corresponding **Atom** object by calling its constructor may be
2 generated by the rule compiler in the hosting language. Such a factory function
3 may handle any of the transformations (e.g., accessing a data field of an object;
4 calculations) from the selected rule's parameters to the **Atom** constructor.

5 **[0068]** The factory function may be added to the rule lookup table at 329. In
6 one embodiment, the factory function and the selected rule's **Rule** instance may
7 have a 1 to 1 association. Accordingly, in one implementation, the factory
8 function's name (or pointer) may be stored with the selected rule's **Rule** instance
9 (e.g., as a data field of a **Rule** object; as part of a [list of **Rule** instances, factory
10 function name] tuple keyed by a rule's identification tuple). In some alternative
11 implementations, the factory function's name (or pointer) may instead be stored
12 in a separate lookup table (e.g., keyed by a rule's identification tuple). In some
13 alternative implementations, in lieu of an explicit look up table of rule data
14 structure, code that calls the appropriate lookup function as specified by the rules
15 may be automatically generated, and a table of function pointers may be created.

16 **[0069]** Data types of the hosting language referenced by the selected rule may
17 be determined at 333. For example, with regard to rule 142, "Int" data type of the
18 hosting language is referenced by this rule. In one implementation, the rule
19 declaration of the selected rule may be parsed to determine the set of data types
20 of the hosting language referenced by the selected rule. For example, this may be
21 done at compile time for compiled languages such as C/C++. In another
22 implementation, run time type inheritance information may be determined (e.g.,

1 for hosting languages that support reflection). For example, this may be done at
2 run time for languages such as Java and .NET.

3 **[0070]** The determined data types may be added to an inheritance tree at 337.
4 For example, the inheritance tree may track data types of the hosting language
5 referenced by the rules in the set of specified rules. In one implementation, the
6 inheritance tree may be a single inheritance-based tree, where each type has a
7 single parent type. For host languages that support reflection, such as .NET, the
8 inheritance tree may be constructed at the run time. For static host languages,
9 such as C++, the inheritance tree may be constructed by parsing the source code,
10 or by relying on a special data type that tracks the inheritance hierarchy, or by
11 using a base C++ data type to automatically track inheritance.

12 **[0071]** The created data structures (e.g., rule data structure instances, the rule
13 lookup table, the inheritance tree) may be stored (e.g., in a data store) at 341. In
14 one implementation, the created data structures may be stored as a shared library
15 at compile time. In another implementation, the created data structures may be
16 stored in memory at run time. Some modern programming languages (e.g., Julia)
17 support run time multiple dispatch. In such host languages, rules may be
18 implemented by generating code that calls the right Atoms as generic methods. In
19 such an implementation, multiple dispatch capability of the host language may be
20 utilized instead of keeping an explicit type inheritance hierarchy.

21 **[0072]** FIGURE 4 shows a logic flow diagram illustrating a logical dependency
22 graph generating (LDGG) component in one embodiment of the DSGRCE. In

Figure 4, an LDG generation request may be obtained at 401. For example, the LDG generation request may be obtained as a result of a user's request to perform a specified calculation using the set of specified rules. It is to be understood that while the below description discusses generating the LDG for a single output rule, the LDGG component may also be utilized with multiple output rules by utilizing elements 405 through 465 for each output rule.

[0073] A specified output rule and rule parameters may be determined at 405.

In one embodiment, the user may provide a list of desired output represented as a set of output rules with concrete parameterization, such as "series.fib(10)". In one implementation, a distributed system execution request may be parsed to determine the specified output rule and/or rule parameters (e.g., based on the value of the calculation_specification field).

[0074] A set of matching rules in the rule lookup table (e.g., corresponding to the specified output rule; corresponding to a selected precedent rule) may be determined at 409. In one implementation, the current rule's (e.g., the specified output rule's; a selected precedent rule's) identification tuple (e.g., "[series, fib]" tuple) may be determined and used to look up a list of matching **Rule** instances from the lookup table (e.g., corresponding to "fib(n::Int)", "fib(n::Int(0))", "fib(n::Int(1))").

[0075] A determination may be made at 413 whether any matching rules have been found. If no matching rules have been found, the DSGRCE may throw an error at 417. For example, the DSGRCE may inform the user that the specified

1 output rule is invalid. In another example, the DSGRCE may inform the user that
2 a selected precedent rule is invalid.

3 **[0076]** If a single matching rule has been found, and the number and types
4 (and/or values) of the run time rule parameters match those in the rule
5 declaration, the matched rule may be selected at 421. A logical dependency graph
6 node identifier for the selected matched rule may be determined at 441. In one
7 implementation, the LDG node identifier for the selected matched rule may be
8 determined as a node identification tuple (e.g., the triplet **[namespace, rulename,**
9 **ruleparameters]**). In another implementation, the LDG node identifier for the
10 selected matched rule may be determined as a hash value of the node
11 identification tuple (e.g., a hash value of the triplet **[namespace, rulename,**
12 **ruleparameters]**).

13 **[0077]** If multiple matching rules have been found, an inheritance distance for
14 each matched rule may be computed at 425. In one embodiment, an inheritance
15 distance for a matched rule may be computed as the sum of the distances between
16 the actual run time type of each rule parameter and the type declared in the
17 matched rule via DSL. In one implementation, a run time parameter that matches
18 the exact type in the rule DSL may be assigned a distance of 0, a run time
19 parameter that has a single level of inheritance may be assigned a distance of 1,
20 etc., while a run time parameter with an exact value match may have a distance
21 of -1. For example, with regard to “series.fib(10)”, the inheritance distance for
22 “fib(n::Int)” may be computed as 0, while “fib(n::Int(0))” and “fib(n::Int(1))” are not

1 matched. In another example, with regard to “series.fib(1)”, the inheritance
2 distance for “fib(n::Int(1))” may be computed as -1, the inheritance distance for
3 “fib(n::Int)” may be computed as 0, while “fib(n::Int(0))” is not matched. In an
4 alternative implementation, the rule matching may be implemented based on the
5 run time multiple dispatch, if the host language supports such features.

6 **[0078]** A determination may be made at 429 whether there is a single matching
7 rule with the minimum inheritance distance. If so, the matching rule with the
8 minimum inheritance distance may be selected at 433. If not, an error may be
9 thrown or a matching rule may be selected using a tie breaker technique (e.g.,
10 based on position of parameters) at 437. In one implementation, a matching rule
11 may be selected using the tie breaker technique based on position of parameters
12 (e.g., the first parameter is the most important for selection, the second parameter
13 is the second most important for selection, etc.). In another implementation, a
14 matching rule may be selected using the tie breaker technique based on types (e.g.,
15 different types of the hosting language may have different priorities with regard
16 to the selection). A logical dependency graph node identifier for the selected
17 matched rule may be determined at 441 as discussed above.

18 **[0079]** A determination may be made at 445 whether an LDG node with the
19 determined LDG node identifier already exists in the LDG. If not, an LDG node
20 for the selected matched rule (e.g., with the **Rule** instance corresponding to the
21 selected matched rule) may be added to the LDG (e.g., implemented as a DAG)

with the determined LDG node identifier at 449. For example, an LDG node may be structured as follows:

```
//additional information may be looked up using a hash value of
//the node identification tuple
struct LogicalGraphNode
    String ns; //namespace
    String name; //rule name
    Vector parameters; //list of parameters of concrete values
end
```

If the selected matched rule is a precedent to an existing node (e.g., this would be the case when processing precedent rules, but not when processing the specified output rule), an LDG edge may be added between the newly added LDG node and the existing dependent LDG node. For example, with regard to “series.fib(10)”, an LDG node with node identification tuple “[series, fib, 10]” may be added. In another example, with regard to “series.fib(9)”, an LDG node with node identification tuple “[series, fib, 9]” may be added, and an LDG edge may be added between the newly added LDG node and the existing dependent node “[series, fib, 10]”. In an alternative implementation, if an LDG node is new, the node’s precedents may be recursively added before adding the node itself (e.g., for each precedent an edge may be added to the graph). For example, with regard to “series.fib(10)”, after adding LDG nodes with node identification tuples “[series, fib, 9]” and “[series, fib, 8]” (e.g., recursively, LDG nodes with node identification tuples “[series, fib, 7]” and “[series, fib, 6]” are added before adding “[series, fib, 8]”), an LDG node with node identification tuple “[series, fib, 10]” may be added

1 and LDG edges may be added between the newly added LDG node and its
2 precedent nodes “[series, fib, 9]” and “[series, fib, 8]”.

3 **[0080]** Precedent rules for the selected matched rule may be determined at 453.
4 In one implementation, the rule data structure, from the rule lookup table,
5 corresponding to the selected matched rule may be queried to determine the
6 precedent rules. For example, with regard to “series.fib(10)”, the precedent rules
7 may be “series.fib(9)” and “series.fib(8)”. In another example, with regard to
8 “series.fib(9)”, the precedent rules may be “series.fib(8)” and “series.fib(7)”. In
9 another example, with regard to “series.fib(1)”, there may not be any precedent
10 rules.

11 **[0081]** A determination may be made at 457 whether there remain precedent
12 rules for the selected matched rule (e.g., a calling dependent rule) to process. In
13 one implementation, each of the determined precedent rules may be processed. If
14 there remain precedent rules to process, the next precedent rule may be selected
15 at 461. The selected precedent rule may be processed as discussed above starting
16 with element 409. Thus, the LDGG component may construct a global logical
17 dependency graph by recursively following and adding precedent rules as new
18 nodes to the LDG, starting from the output node corresponding to the specified
19 output rule.

20 **[0082]** If there do not remain precedent rules to process, a determination may
21 be made at 465 whether the current rule is the specified output rule. If the current
22 rule is not the specified output rule, the LDGG component may return to

1 processing precedent rules of the current rule's calling dependent rule at 475. For
2 example, with regard to "series.fib(9)", once precedent rules "series.fib(8)" and
3 "series.fib(7)" have been processed, the LDGG component may return to
4 processing precedent rule "series.fib(8)" of the calling dependent rule
5 "series.fib(10)". If the current rule is the specified output rule, the LDGG
6 component may provide the generated LDG at 481. For example, a reference to
7 the generated LDG (e.g., stored in memory) may be provided.

8 **[0083]** If it is determined at 445 that the LDG node with the determined LDG
9 node identifier already exists in the LDG, an LDG edge may be added between the
10 existing LDG node with the determined LDG node identifier and the LDG node
11 associated with the calling dependent rule at 471. For example, with regard to
12 processing "series.fib(8)" with recursive dependency "series.fib(10)", if the LDG
13 node with node identification tuple "[series, fib, 8]" was already added to the LDG
14 while processing "series.fib(9)", an LDG edge may be added between the LDG node
15 with node identification tuple "[series, fib, 8]" and the calling dependent rule node
16 "[series, fib, 10]". The LDGG component may return to processing precedent rules
17 of the current rule's calling dependent rule starting at 475 as discussed above.

18 **[0084]** FIGURE 5 shows a logic flow diagram illustrating a worker subgraph
19 distribution (WSD) component in one embodiment of the DSGRCE. In Figure 5, a
20 worker subgraph distribution request may be obtained at 501. For example, the
21 worker subgraph distribution request may be obtained as a result of the user's

1 request to perform the specified calculation using the set of specified rules (e.g.,
2 once the logical dependency graph is generated).

3 **[0085]** The logical dependency graph may be obtained at 505. In one
4 implementation, the generated LDG may be provided by the LDGG component.
5 For example, a reference to the generated LDG (e.g., stored in memory) may be
6 obtained.

7 **[0086]** A determination may be made at 509 regarding what execution
8 complexity gauge to use. In one embodiment, the number of nodes may be used to
9 gauge execution complexity of the LDG. If so, the number of nodes in the LDG
10 may be determined at 513. In one implementation, a depth-first search (DFS) or
11 a breadth-first search (BFS) through the LDG may be utilized to determine the
12 number of nodes. In another embodiment, tensor sizes may be used to gauge
13 execution complexity of the LDG. If so, the sum of output tensor sizes of LDG
14 nodes may be determined at 517. In one implementation, a DFS or a BFS through
15 the LDG may be utilized to determine the sum of output tensor sizes. In another
16 embodiment, network data transfer size may be used to gauge execution
17 complexity of the LDG. If so, estimated network data transfer size associated with
18 the LDG may be determined at 519. In one implementation, statistics collected
19 from previous runs may be utilized to estimate CPU utilization or the network
20 data transfer size (e.g., based on an average of sizes from previous runs). In some
21 embodiments, the execution complexity gauge may be a weighted average of

multiple metrics, such as the number of nodes, CPU usage, the network data transfer size, the tensor output size, and/or the like.

[0087] A determination may be made at 521 whether the number of available workers (e.g., corresponding to worker processes) is fixed or dynamic. In one implementation, the distributed system execution request may be parsed to make this determination (e.g., based on the value of the type field). If the number of available workers is fixed, the specified number of available workers may be determined at 525. In one implementation, the distributed system execution request may be parsed to determine the specified number of available workers (e.g., based on the value of the number_of_available_workers field). For example, 6 workers may be available.

[0088] If the number of available workers is dynamic, scaling parameters for calculating the number of workers to utilize may be determined at 529. In various implementations, the scaling parameters may include one or more of: target processing time, target available memory, target cost, and/or the like. The number of workers to utilize may be calculated at 533. In one implementation, the execution complexity (e.g., the number of nodes in the LDG, the sum of output tensor sizes of LDG nodes, the network data transfer size) may be divided by the average worker execution capability (e.g., determined based on previous experience, determined based on worker hardware properties) with regard to the scaling parameters to calculate the number of workers to utilize. For example, if the execution complexity of the LDG is 80,000 nodes, the scaling parameters

indicate that the user wishes to complete the specified calculation in under one second, and the average worker execution capability is 15,000 nodes per second, the number of workers to utilize may be calculated as follows:

$$1 \text{ second} = 80,000 \text{ nodes} / 15,000 \frac{\text{nodes}}{\text{second}} / \# \text{ of Workers}$$

$$\# \text{ of Workers} = \left\lceil 80,000 \text{ nodes} / 15,000 \frac{\text{nodes}}{\text{second}} / 1 \text{ second} \right\rceil$$

$$\# \text{ of Workers} = 6$$

[0089] In another example, if the execution complexity of the LDG is sum of output tensor sizes of 120,000 (e.g., with each corresponding to utilizing 0.5MB of memory on average), the scaling parameters indicate that the user wishes to complete the specified calculation without running out of memory, and the average worker execution capability is 16GB of memory, the number of workers to utilize may be calculated as follows:

$$\text{sum of output tensor sizes of } 120,000 * 0.5\text{MB per output tensor size} = 60\text{GB utilized}$$

$$\# \text{ of Workers} = \left\lceil 60\text{GB} / 16 \frac{\text{GB}}{\text{worker}} \right\rceil$$

$$\# \text{ of Workers} = 4$$

[0090] Execution processing capabilities of workers may be determined at 537. For example, the execution processing capabilities may include available hardware properties (e.g., actual hardware properties, hardware properties reported by a cloud-based virtual machine) such as CPU type, number of cores per CPU, available memory, network communication speed, and/or the like. In one implementation, execution processing capabilities of workers (e.g., executing on a

1 number of heterogeneous computers) may be determined from a hardware
2 configuration setting (e.g., a configuration file that lists hardware properties of
3 available computers). In another implementation, execution processing
4 capabilities of workers (e.g., homogeneous workers executing in a cloud computing
5 environment) may be determined from a workers configuration setting (e.g., a
6 configuration setting that specifies hardware properties for each worker). It is to
7 be understood that, in some implementations, multiple workers may execute on a
8 computer system (e.g., on a physical machine or on a virtual machine).

9 **[0091]** The LDG may be divided into subgraphs based on the number and/or
10 execution processing capabilities of workers and execution complexity of LDG at
11 541. In one embodiment, the LDG may be divided into subgraphs such that each
12 worker is assigned to process a portion of the LDG commensurate with the
13 respective worker's execution processing capabilities. In one implementation, if
14 the workers have homogeneous execution processing capabilities, the LDG may
15 be divided into subgraphs having similar execution complexity. For example, if
16 the execution complexity of the LDG is 80,000 nodes and there are 6 workers, the
17 LDG may be divided into 6 subgraphs having 13,000, 13,000, 13,000, 13,000,
18 14,000, and 14,000 nodes. In another implementation, if the workers have
19 heterogeneous execution processing capabilities, the LDG may be divided into
20 subgraphs that correspond to the available execution processing capabilities of
21 each worker. For example, if the execution complexity of the LDG is sum of output
22 tensor sizes of 120,000 utilizing 60GB of memory and there are 4 workers with

1 24GB, 24GB, 8GB, and 8GB of available memory, the LDG may be divided into 4
2 subgraphs having 45,000, 45,000, 15,000, and 15,000 sum of output tensor sizes.

3 **[0092]** A determination may be made at 545 whether to reduce communications
4 overhead to optimize processing. For example, in some cases, communications
5 overhead associated with a node calculation (e.g., sending output of the node
6 calculation from one worker to another worker) may be more costly than
7 duplicating the node calculation (e.g., at the other worker). In one implementation,
8 the distributed system execution request may be parsed to determine whether to
9 reduce the communications overhead (e.g., based on the value of the
10 communications_overhead field).

11 **[0093]** If it is determined that the communications overhead should be reduced,
12 hot nodes with large communications overhead may be determined at 549. In one
13 implementation, a node may be classified as a hot node if the node exceeds a
14 specified data transfer size threshold. For example, data transfer size may be
15 calculated as output size (e.g., output tensor size) multiplied by the number of
16 remote workers that depend on the output (e.g., the number of workers that would
17 have to receive the output over a network). The determined hot nodes may be
18 duplicated on subgraphs of other workers at 553. In one implementation, if a hot
19 node's output is a remote precedence for a subgraph node, the hot node may be
20 added to the subgraph associated with the subgraph node. It is to be understood
21 that the duplication of hot nodes may be applied recursively, resulting in a portion
22 of the entire graph being duplicated in multiple workers, while providing faster

1 computation speed of the specified calculation due to reduced communications
2 overhead.

3 **[0094]** A determination may be made at 557 whether there remain workers to
4 process. In one implementation, each of the workers to be utilized may be
5 processed. If there remain workers to process, the next worker may be selected for
6 processing at 561.

7 **[0095]** A subgraph for the selected worker may be determined at 565. In one
8 implementation, if the workers have homogeneous execution processing
9 capabilities, any unassigned subgraph may be assigned to the selected worker
10 (e.g., randomly). In another implementation, if the workers have heterogeneous
11 execution processing capabilities, a subgraph that corresponds to the available
12 execution processing capabilities of the selected worker may be determined and
13 assigned to the worker. In one embodiment, the global allocation of nodes
14 (subgraphs) to workers (e.g., determined centrally by the WSD component) may
15 be broadcasted to each worker, so that each worker knows the global allocation of
16 nodes to workers. Based on that information, any worker may determine if a node
17 is local or remote, and the remote worker that owns any nodes.

18 **[0096]** Remote precedences for the selected worker may be determined at 569.
19 For example, a remote precedence may be an output of a node in the subgraph of
20 another worker that is an input to a node in the subgraph assigned to the selected
21 worker. In one implementation, the subgraph assigned to the selected worker may
22 be analyzed to determine whether each input to each node in the subgraph has a

1 remote precedence (e.g., by comparing against the LDG) and/or to determine the
2 identifier (e.g., remote worker identifier and/or remote node identifier) of the
3 remote precedence.

4 **[0097]** Remote dependencies for the selected worker may be determined at 573.

5 For example, a remote dependency may be an input to a node in the subgraph of
6 another worker that is an output of a node in the subgraph assigned to the selected
7 worker. In one implementation, the subgraph assigned to the selected worker may
8 be analyzed to determine whether each output of each node in the subgraph has
9 a remote dependency (e.g., by comparing against the LDG) and/or to determine
10 the identifier (e.g., remote worker identifier and/or remote node identifier) of the
11 remote dependency.

12 **[0098]** A worker process for the selected worker may be instantiated (e.g.,
13 including the LDG data structure) at 577. For example, data such as the subgraph
14 assigned to the selected worker, information regarding the remote precedences,
15 information regarding the remote dependencies, a communication mode (e.g.,
16 worker to worker, shared cache), and/or the like may be provided to the
17 instantiated worker process. In one implementation, an instantiation request may
18 be sent to a computer system utilized to execute the worker process (e.g., to a
19 physical machine, to a virtual machine) to instantiate the worker process. In some
20 embodiments, the worker process may be a service that is continuously running.

21 **[0099]** FIGURE 6 shows a logic flow diagram illustrating a worker
22 instantiation (WI) component in one embodiment of the DSGRCE. In Figure 6, an

1 instantiation request may be obtained at 601. For example, the instantiation
2 request may be obtained as a result of a manager process sending the
3 instantiation request to instantiate a worker process.

4 **[00100]** A specified LDG subgraph for the worker process may be determined at
5 605. In one implementation, the instantiation request may be parsed to determine
6 the specified LDG subgraph for the worker process (e.g., based on the value of the
7 LDG_subgraph field).

8 **[00101]** A determination may be made at 609 whether there remain subgraph
9 nodes to process. In one implementation, each of the subgraph nodes in the
10 specified LDG subgraph may be processed. If there remain subgraph nodes to
11 process, the next node may be selected for processing at 613.

12 **[00102]** Remote precedences for the selected subgraph node may be determined
13 at 617. In one implementation, the instantiation request may be parsed to
14 determine the remote precedences (e.g., based on the value of the precedences field
15 associated with a node identifier of the selected subgraph node). For example, a
16 remote worker process identifier and/or a remote node identifier associated with
17 each remote precedence (e.g., input) for the selected subgraph node may be
18 determined. In another implementation, the manager process may provide the
19 global allocation of nodes (subgraphs) to workers (e.g., determined centrally by the
20 WSD component), so that the worker process may determine whether the selected
21 subgraph node has remote precedences. For example, a remote worker process
22 identifier and/or a remote node identifier associated with each remote precedence

(e.g., input) for the selected subgraph node may be determined from the global allocation of nodes.

[00103] Remote dependencies for the selected node may be determined at 621.

In one implementation, the instantiation request may be parsed to determine the remote dependencies (e.g., based on the value of the dependencies field associated with the node identifier of the selected subgraph node). For example, a remote worker process identifier and/or a remote node identifier associated with each remote dependency (e.g., output) for the selected subgraph node may be determined. In another implementation, the manager process may provide the global allocation of nodes (subgraphs) to workers (e.g., determined centrally by the WSD component), so that the worker process may determine whether the selected subgraph node has remote dependencies. For example, a remote worker process identifier and/or a remote node identifier associated with each remote dependency (e.g., output) for the selected subgraph node may be determined from the global allocation of nodes.

[00104] A determination may be made at 625 regarding what communication mode to use. In one embodiment, a worker to worker communication mode may be used. If so, network information of the remote precedences may be determined at 631. For example, the network information may include an IP address and/or port of each remote worker process whose output is used as an input by the selected subgraph node. In one implementation, the instantiation request may be parsed to determine the network information of the remote precedences (e.g.,

1 based on the value of the worker_IP_address field associated with each
2 corresponding remote worker process identifier).

3 **[00105]** The worker process may subscribe to notifications from the remote
4 precedences at 635. For example, a notification from a remote precedence worker
5 process may include output of a remote precedence node that is used as an input
6 by the selected subgraph node (e.g., the notification may be sent after the remote
7 precedence worker process finishes a node calculation associated with the remote
8 precedence node). In one implementation, the worker process may send a
9 subscription request to the remote precedence worker process (e.g., using the
10 network information of the remote precedence worker process) specifying the node
11 identifier of the remote precedence node whose output the worker process wishes
12 to obtain.

13 **[00106]** The worker process may register contacting dependencies for
14 notifications regarding the selected subgraph node at 639 (e.g., this may be done
15 in a separate thread any time a subscription request is received). For example, a
16 notification from the worker process may include output of the selected subgraph
17 node that is used as an input by a remote dependency node of the contacting
18 remote dependency worker process (e.g., the notification may be sent after the
19 worker process finishes a node calculation associated with the selected subgraph
20 node). In one implementation, the worker process may verify that the contacting
21 remote dependency worker process is authorized to register by checking that the

1 contacting remote dependency worker process is one of the determined remote
2 dependencies for the selected subgraph node.

3 **[00107]** In some alternative implementations, instead of utilizing subscription
4 requests to register remote worker processes for notifications (e.g., utilized when
5 network information of remote dependencies is not available in the instantiation
6 request), the determined remote dependencies for the selected subgraph node may
7 be automatically registered for notifications (e.g., utilized when network
8 information of remote dependencies is available in the instantiation request)
9 without the use of requests to register from the determined remote dependencies.

10 **[00108]** In another embodiment, a shared cache communication mode may be
11 used. If so, network information of a shared cache may be determined at 641. For
12 example, the network information may include an IP address and/or port of the
13 shared cache. In one implementation, the instantiation request may be parsed to
14 determine the network information of the shared cache (e.g., based on the value
15 of the `shared_cache_network_configuration` field).

16 **[00109]** The worker process may subscribe to shared cache notifications from
17 remote precedences at 645. For example, a notification from the shared cache may
18 include output of a remote precedence node from a remote precedence worker
19 process that is used as an input by the selected subgraph node (e.g., the
20 notification may be sent after the remote precedence worker process finishes a
21 node calculation associated with the remote precedence node and notifies the
22 shared cache). In one implementation, the worker process may send a subscription

1 request to the shared cache (e.g., using the network information of the shared
2 cache) specifying the remote worker process identifier of the remote precedence
3 worker process and/or the remote node identifier of the remote precedence node
4 whose output the worker process wishes to obtain from the shared cache. In some
5 alternative embodiments, instead of using a subscription model, the shared cache
6 may be passive (e.g., the shared cache does not actively push data to workers) and
7 the worker process may send a request to the shared cache to retrieve the status
8 and/or value of any data it wishes to obtain (e.g., output of a remote precedence
9 node from a remote precedence worker process that is used as an input by the
10 selected subgraph node).

11 **[00110]** The worker process may register with the shared cache to notify remote
12 dependencies at 649. For example, a notification from the worker process may
13 include output of the selected subgraph node that is used as an input by a remote
14 dependency node. In one implementation, the worker process may send a
15 registration request to the shared cache (e.g., using the network information of
16 the shared cache) specifying the worker process identifier of the worker process
17 and/or the node identifier of the selected subgraph node whose output the worker
18 process will provide to the shared cache. In some alternative embodiments,
19 instead of using a subscription model, the shared cache may be passive (e.g., the
20 shared cache does not actively push data to workers) and the worker process may
21 send a request to the shared cache to provide the status and/or value of any data

1 it wishes to send (e.g., output of the selected subgraph node that is used as an
2 input by a remote dependency node).

3 **[00111]** A rule associated with the selected subgraph node may be determined
4 at 651. In one implementation, the **Rule** instance associated with the selected
5 subgraph node may be determined. The factory function associated with the rule
6 may be called at 655. In one implementation, calling the factory function creates
7 a corresponding **Atom** object by calling its constructor. For example, the input
8 tensor size to the constructor of an **Atom** may be automatically inferred by calling
9 the **dims()** interface of its precedent atoms. In some implementations, if an atom
10 is fully functional and has no internal state (e.g., no write update to the same
11 memory location is possible), a single instance of the atom may be shared for the
12 subgraph instead of duplicating the fully functional atom.

13 **[00112]** An instantiation response may be provided at 659. For example, the
14 instantiation response may be sent to the manager process to inform the manager
15 process that the worker process instantiated successfully.

16 **[00113]** FIGURE 7 shows a logic flow diagram illustrating a graph execution
17 (GE) component in one embodiment of the DSGRCE. In Figure 7, a graph
18 execution request may be obtained at 701. For example, the graph execution
19 request may be obtained as a result of the user's request to perform the specified
20 calculation using the set of specified rules (e.g., once at least some of the worker
21 processes have been instantiated).

1 **[00114]** A requested output for a current run may be determined at 705. For
2 example, the requested output may be the present value of a fixed income portfolio
3 (e.g., provided as input) and/or the present value's sensitivities. In some
4 embodiments, the specified calculation may be repeated multiple times (e.g., an
5 intraday trading system may recalculate the present value of a fixed income
6 portfolio each time an updated quote for a constituent bond is received) with some
7 unmodified inputs (e.g., infrequently changing values of corporate bonds) and
8 some modified inputs (e.g., frequently changing values of equity prices) between
9 subsequent runs.

10 **[00115]** LDG nodes to compute for the requested output may be determined at
11 709. In one embodiment, each of the LDG nodes may be computed. In another
12 embodiment, LDG nodes that depend on inputs that were modified from the
13 preceding run may be computed. For example, affected LDG nodes that use
14 modified inputs may be determined as LDG nodes to compute, and any other LDG
15 nodes that depend on output from the affected LDG nodes (e.g., directly or
16 indirectly) may also be determined as LDG nodes to compute. In one
17 implementation, a DFS or a BFS through the LDG may be utilized to determine
18 the LDG nodes to compute (e.g., starting from the affected LDG nodes).

19 **[00116]** A determination may be made at 713 whether there remain worker
20 processes to process. In one implementation, each of the worker processes may be
21 processed. If there remain worker processes to process, the next worker process
22 may be selected for processing at 717.

1 **[00117]** Subgraph nodes to run for the selected worker process may be
2 determined at 721. In one implementation, the determined LDG nodes to compute
3 that are included in the LDG subgraph associated with the selected worker
4 process may be determined as the subgraph nodes to run for the selected worker
5 process. In an alternative implementation, the subgraph nodes to run may be
6 determined during instantiation at 577.

7 **[00118]** A manager process may notify the selected worker process to initialize
8 the forward (or primary) calculation (e.g., a portion of the specified calculation) at
9 731. In one implementation, the manager process may send a computation request
10 to the selected worker process to instruct the selected worker process to initialize
11 the forward calculation. In some implementations, the forward initialization (e.g.,
12 to reset run status from a previous run, such as whether a node has been executed,
13 or whether a node has received preceding data from another worker) may be a
14 part of the forward calculation request discussed at 741 instead of being a separate
15 request.

16 **[00119]** A determination may be made at 735 whether the forward calculation
17 initialization was completed by the selected worker process. In one
18 implementation, a computation response may be received from the selected
19 worker process indicating that the forward calculation initialization was
20 completed. If the forward calculation initialization was not completed, the GE
21 component may wait at 739. In one implementation, the wait may be non-blocking.
22 For example, the GE component may move on to processing the next worker

1 process and may return to processing the selected worker process at a later time
2 (e.g., after looping through the worker processes, after being notified that the
3 forward calculation initialization was completed). In another example, instead of
4 looping, graph execution may be implemented as part of the event loop of the
5 message handler.

6 **[00120]** If the forward calculation initialization was completed, the manager
7 process may notify the selected worker process to run the forward calculation at
8 741. In one implementation, the manager process may send a computation request
9 to the selected worker process to instruct the selected worker process to run the
10 forward calculation. For example, the computation request may specify subgraph
11 nodes to run for the selected worker process (e.g., if only a portion of the graph
12 should to be updated (e.g., affected by changing inputs)).

13 **[00121]** A determination may be made at 745 whether the forward calculation
14 was completed by the selected worker process. In one implementation, a
15 computation response may be received from the selected worker process indicating
16 that the forward calculation was completed. If the forward calculation was not
17 completed, the GE component may wait at 749. In one implementation, the wait
18 may be non-blocking. For example, the GE component may move on to processing
19 the next worker process and may return to processing the selected worker process
20 at a later time (e.g., after looping through the worker processes, after being
21 notified that the forward calculation was completed). In another example, instead
22 of looping, graph execution may be implemented as part of the event loop of the

1 message handler. It is to be understood that if forward AD calculation is desired,
2 it may be either part of the forward primal step, or a second forward round that is
3 similar to the forward primal calculation.

4 **[00122]** If the forward calculation was completed, the manager process may
5 notify the selected worker process to initialize the backward calculation (e.g.,
6 calculation of sensitivities) at 751. In one implementation, the manager process
7 may send a computation request to the selected worker process to instruct the
8 selected worker process to initialize the backward calculation. In some
9 implementations, the backward initialization (e.g., to reset run status from a
10 previous run, such as whether a node has been executed, or whether a node has
11 received preceding data from another worker) may be a part of the backward
12 calculation request discussed at 761 instead of being a separate request. It is to
13 be understood that, in some embodiments, calculation of sensitivities may not be
14 requested by the user, in which case the backward calculation initialization and/or
15 the backward calculation may be avoided.

16 **[00123]** A determination may be made at 755 whether the backward calculation
17 initialization was completed by the selected worker process. In one
18 implementation, a computation response may be received from the selected
19 worker process indicating that the backward calculation initialization was
20 completed. If the backward calculation initialization was not completed, the GE
21 component may wait at 759. In one implementation, the wait may be non-blocking.
22 For example, the GE component may move on to processing the next worker

1 process and may return to processing the selected worker process at a later time
2 (e.g., after looping through the worker processes, after being notified that the
3 backward calculation initialization was completed). In another example, instead
4 of looping, graph execution may be implemented as part of the event loop of the
5 message handler of network communication.

6 **[00124]** If the backward calculation initialization was completed, the manager
7 process may notify the selected worker process to run the backward calculation at
8 761. In one implementation, the manager process may send a computation request
9 to the selected worker process to instruct the selected worker process to run the
10 backward calculation. For example, the computation request may specify
11 subgraph nodes to run for the selected worker process.

12 **[00125]** A determination may be made at 765 whether the backward calculation
13 was completed by the selected worker process. In one implementation, a
14 computation response may be received from the selected worker process indicating
15 that the backward calculation was completed. If the backward calculation was not
16 completed, the GE component may wait at 769. In one implementation, the wait
17 may be non-blocking. For example, the GE component may move on to processing
18 the next worker process and may return to processing the selected worker process
19 at a later time (e.g., after looping through the worker processes, after being
20 notified that the backward calculation was completed). In another example,
21 instead of looping, graph execution may be implemented as part of the event loop
22 of the message handler.

1 **[00126]** If there do not remain worker processes to process, the manager process
2 may retrieve the requested output at 771. For example, the requested output may
3 be the result of the specified calculation (e.g., including sensitivities). In another
4 example, the requested output may be the result of a calculation by any LDG node
5 (e.g., when the user wishes to view intermediate results by selecting an LDG node
6 of interest to the user using a DSGRCE GUI). In one implementation, the manager
7 process may retrieve the requested output by sending a message to a relevant
8 worker process (e.g., determined based on the LDG subgraph that contains the
9 selected LDG node) to provide results for a specified LDG node.

10 **[00127]** The requested output may be provided at 775. In one implementation,
11 the manager process may send a distributed system execution response to provide
12 the requested output.

13 **[00128]** FIGUREs 8A-C show a logic flow diagram illustrating a worker
14 computation (WC) component in one embodiment of the DSGRCE. In Figure 8A,
15 a computation request may be obtained at 801. For example, the computation
16 request may be obtained as a result of a manager process instructing a worker
17 process to execute a computation.

18 **[00129]** A determination may be made at 805 regarding the request type
19 specified in the computation request. If the request type is “initialize forward”,
20 allocated memory from previous runs may be cleared and/or released by the
21 worker process at 811. In one implementation, instantiated objects from previous

runs may be deleted. Variables may be initialized at 815. In one implementation, objects may be instantiated and/or initialized to their default values.

[00130] If the request type is “calculate forward”, the specified LDG subgraph for the worker process may be determined at 821. In one implementation, a list of node IDs (e.g., that stores the LDG subgraph for the worker process specified in an instantiation request) may be accessed to determine the specified LDG subgraph. Results of the forward calculation for the LDG subgraph may be computed at 825. In one implementation, subgraph nodes to run for the worker process (e.g., specified by the manager process) may be processed to compute the results of the forward calculation. See Figure 8B for additional details regarding computing the results of the forward calculation.

[00131] If the request type is “initialize backward”, the final Jacobian may be initialized to the identity tensor at 831 and intermediate Jacobians may be initialized to the zero tensor at 835. In one implementation, the computation request may include the identity of ultimate output $\vec{u}$ and the sensitivity set (e.g., a list of nodes for which sensitivities should be computed: the $\vec{x}$ s in $\frac{\partial \vec{u}}{\partial \vec{x}}$ where each $\vec{x}_i$ is a node that $\vec{u}$ depends on and whose sensitivities are requested by the user. Upon receiving the computation request the worker process may initialize the final Jacobian to the identity tensor $\frac{\partial \vec{u}}{\partial \vec{u}}$ and the intermediate Jacobians to the 0 tensor. In some implementations, the memory utilized for computing the intermediate Jacobians may be preallocated at this stage to improve memory efficiency.

1 **[00132]** If the request type is “calculate backward”, the specified LDG subgraph
2 for the worker process may be determined at 841. In one implementation, a list of
3 node IDs (e.g., that stores the LDG subgraph for the worker process specified in
4 an instantiation request) may be accessed to determine the specified LDG
5 subgraph. Sensitivities for the LDG subgraph may be computed at 845. In one
6 implementation, the worker process may execute the LDG subgraph backwards,
7 from the ultimate output to input, by calling the **back()** method on each **Atom**
8 object of each node, and stops when the sensitivity set $\bar{x}_i$ has been executed (e.g.,
9 upon completion the $\frac{\partial \bar{u}}{\partial x_i}$ are computed). In some implementations, when an
10 intermediate result (e.g., intermediate tensor and/or gradient result) is no longer
11 used (e.g., determined based on the LDG subgraph), memory allocated for the
12 intermediate result may be reused for other nodes to improve memory efficiency.
13 See Figure 8C for additional details regarding computing the results of the
14 backward calculation.

15 **[00133]** In some implementations, a forward AD calculation may be used, which
16 works similarly to the backward AD calculation described above, but with
17 different initialization and calculation order.

18 **[00134]** A computation response may be provided at 851. In one implementation,
19 the worker process may send a status indicator and/or the requested computation
20 results to the manager process via the computation response.

21 **[00135]** FIGURE 8B shows additional details regarding computing the results
22 of the forward calculation. In Figure 8B, a determination may be made at 862

whether there remain subgraph nodes to process. In one implementation, each of the specified subgraph nodes to run in the specified LDG subgraph may be processed. If there remain subgraph nodes to process, nodes in the ready set may be determined at 864. In one implementation, the ready set may include unprocessed subgraph nodes that are ready to run, which may include unprocessed subgraph nodes that do not have precedences and unprocessed subgraph nodes whose precedences have already been processed (e.g., by the worker process for local precedences; by another worker process for remote precedences). The next node in the ready set may be selected for processing at 866.

[00136] A determination may be made at 868 whether the selected node has remote precedences. If so, the preceding inputs (e.g., calculation outputs of remote nodes) may be determined from corresponding remote worker processes at 870. In one implementation, the preceding inputs may be obtained directly from the remote worker processes. In another implementation, the preceding inputs may be obtained from a shared cache.

[00137] Results for the selected node may be calculated at 872. In one implementation, the **fwd()** method of the **Atom** object associated with the selected node may be called with any local precedences (e.g., provided as input; previously calculated by the worker process for one or more already processed nodes) and/or remote precedences (e.g., obtained from one or more remote worker processes) provided as input.

1 **[00138]** A determination may be made at 874 whether the selected node has
2 remote dependencies. If so, the calculation results may be provided for remote
3 worker processes at 876. In one implementation, the calculation results may be
4 sent directly to the remote worker processes. In another implementation, the
5 calculation results may be sent to a shared cache.

6 **[00139]** FIGURE 8C shows additional details regarding computing the results
7 of the backward calculation. It is to be understood that precedent and dependent
8 nodes are defined by the direction of data flow. Accordingly, precedent and
9 dependent nodes may reverse between forward and backward runs, as their
10 direction of data flow is different. In Figure 8C, a determination may be made at
11 882 whether there remain subgraph nodes to process. In one implementation, each
12 of the specified subgraph nodes to run in the specified LDG subgraph may be
13 processed. If there remain subgraph nodes to process, nodes in the ready set may
14 be determined at 884. In one implementation, the ready set may include
15 unprocessed subgraph nodes that are ready to run, which may include
16 unprocessed subgraph nodes that do not have precedences and unprocessed
17 subgraph nodes whose precedences have already been processed (e.g., by the
18 worker process for local precedences; by another worker process for remote
19 precedences). The next node in the ready set may be selected for processing at 886.

20 **[00140]** A determination may be made at 888 whether the selected node has
21 remote precedences. If so, the preceding inputs (e.g., sensitivities outputs of
22 remote nodes) may be determined from corresponding remote worker processes at

890. In one implementation, the preceding inputs may be obtained directly from the remote worker processes. In another implementation, the preceding inputs may be obtained from a shared cache.

[00141] Sensitivities for the selected node may be calculated at 892. In one implementation, the **back()** method of the **Atom** object associated with the selected node may be called with any local precedences (e.g., provided as input; previously calculated by the worker process for one or more already processed nodes) and/or remote precedences (e.g., obtained from one or more remote worker processes) provided as input. In another implementation (e.g., when a forward AD calculation is used), the **fwdad()** method of the **Atom** object associated with the selected node may be called.

[00142] A determination may be made at 894 whether the selected node has remote dependencies. If so, the sensitivities may be provided for remote worker processes at 896. In one implementation, the sensitivities may be sent directly to the remote worker processes. In another implementation, the sensitivities may be sent to a shared cache.

[00143] FIGURE 9 shows a screen shot diagram illustrating user interface features in one embodiment of the DSGRCE. In Figure 9, an exemplary user interface (e.g., for a mobile device, for a website) showing a logical dependency graph 901 for calculating a Fibonacci series is illustrated. The LDG corresponds to using rule declarations 130 with scalar “10” as input. Panel 905 shows information regarding the selected node 915 (e.g., “series.fib(10)”) of the LDG. The

1 final results (e.g., 55) associated with the selected node are displayed as a bar
2 chart 910. The **Atom** object associated with the selected node is shown at 920. Rule
3 parameters associated with the selected node are shown at 925. Precedence nodes
4 for the selected node are shown at 930. Dependency nodes for the selected node
5 are shown at 935. Input and output dimensions for the selected node are shown
6 at 940. The source code (e.g., the rule declaration; related metadata such as
7 developer name, documentation, source code location) associated with the selected
8 node is shown at 945.

9 **[00144]** FIGURE 10 shows a screen shot diagram illustrating user interface
10 features in one embodiment of the DSGRCE. In Figure 10, an exemplary user
11 interface (e.g., for a mobile device, for a website) showing two logical dependency
12 graphs is illustrated. LDG 1001 corresponds to LDG 901. LDG 1005 is the result
13 of rule cloning and override corresponding to using rule declarations 160 with
14 “series_dev.fib(10)” as user specified output. Figure 10 shows that calculation
15 results associated with “series” declarations are not lost, and may be compared
16 with results associated with “series_dev” declarations after the rules have
17 changed.

18 **[00145]** FIGURE 11 shows a screen shot diagram illustrating user interface
19 features in one embodiment of the DSGRCE. In Figure 11, an exemplary user
20 interface (e.g., for a mobile device, for a website) showing a rule dependency graph
21 1101 is illustrated. In one implementation, the rule dependency graph (RDG) may
22 be generated by traversing the logical dependency graph (e.g., starting with the

LDG output node corresponding to the specified output rule (e.g., series_dev.fib(10))). An RDG node corresponding to the **Rule** instance associated with the LDG output node may be added to the rule dependency graph. For example, an RDG node may be structured as follows:

```

5      //additional information may be looked up using a hash value of
6      //the node identification tuple
7      struct RuleGraphNode
8          String ns; //namespace
9          String name; //rule name
10         Vector signature; //list of types (or types(value) for
11                             //value based polymorphism) of the
12                             //rule parameter (e.g., Double, Int(3))
13     end
14

```

Precedent LDG nodes of the LDG output node may be recursively traversed, and a **Rule** instance associated with each respective LDG precedent node may be determined. If an RDG node corresponding to a **Rule** instance associated with a respective LDG precedent node does not yet exist in the rule dependency graph, such RDG node may be added to the rule dependency graph and an RDG edge between such RDG node and its dependent RDG node may be added. If such RDG node already exists, an RDG edge between such RDG node and its dependent RDG node may be added (e.g., an RDG edge may be between two different RDG nodes, or between an RDG node and itself). For example, an RDG node may provide information regarding the associated **Rule** instance (e.g., the rule declaration; related metadata such as developer name, documentation, source code location),

LDG nodes that utilize the associated **Rule** instance, precedent and/or dependent RDG nodes (e.g., via RDG edges data fields), and/or the like. In one implementation, a user may utilize the pull-down menu 1105 to switch the display between the LDG (calculation graph) and the rule dependency graph. The rule dependency graph shows the logic lineage of the calculation, and clearly identifies that there is only a single rule different between namespace “series” and namespace “series_dev”. The rule dependency graph shows that rule cloning is effectively copy-on-write (e.g., when a set of rules is first cloned, the cloned rules refer to the identical rules from the original set, until they are explicitly changed, as with regard to rule “series_dev.fib(2)”. The rule dependency graph may be utilized for model validation and audit.

[00146] FIGURE 12 shows a screen shot diagram illustrating user interface features in one embodiment of the DSGRCE. In Figure 12, an exemplary user interface (e.g., for a mobile device, for a website) showing a logical dependency graph 1201 for calculating a Fibonacci series using packages is illustrated. The LDG corresponds to using rule declarations 170 with two random vectors of length 3 as inputs. Bar chart 1205 shows one of the random inputs.

[00147] FIGURE 13 shows a screen shot diagram illustrating user interface features in one embodiment of the DSGRCE. In Figure 13, an exemplary user interface (e.g., for a mobile device, for a website) showing a logical dependency graph 1301 that provides details regarding how LDG node “pack.fib(3)” 1210 is computed is illustrated. In one implementation, a user may drill down (e.g., by

1 double clicking on the LDG node) to the inner graph of the packaged atom
2 “pack.fib(3)” to see the details.

3 **[00148]** FIGURE 14 shows a screen shot diagram illustrating user interface
4 features in one embodiment of the DSGRCE. In Figure 14, an exemplary user
5 interface (e.g., for a mobile device, for a website) for showing sensitivities is
6 illustrated. Sensitivities 1401 to the two random vectors associated with the
7 logical dependency graph 1201 (e.g., using algorithmic differentiation to obtain
8 results) are shown. The sensitivity to each of the two input vectors is a Jacobian
9 matrix of 3x3, and the sensitivities are displayed as two columns of line diagrams.

10 **[00149]** FIGURE 15 shows a screen shot diagram illustrating user interface
11 features in one embodiment of the DSGRCE. In Figure 15, an exemplary user
12 interface (e.g., for a mobile device, for a website) showing status of a DSGRCE
13 generated distributed system is illustrated. Status 1501 associated with
14 calculating pricing results for a large portfolio with a million trades on Amazon
15 Web Services (AWS) cloud using 6 workers is shown. The user interface shows
16 details such as the network address, computation progress, number of nodes in
17 distributed LDG subgraph, CPU utilization, memory utilization, messages sent
18 for data exchanges, messages received for data exchanges, and/or the like for each
19 of the 6 workers.

20 **[00150]** FIGURE 16 shows a screen shot diagram illustrating user interface
21 features in one embodiment of the DSGRCE. In Figure 16, an exemplary user
22 interface (e.g., for a mobile device, for a website) showing a logical dependency

graph 1601 for calculating the pricing results for the large portfolio and the top level results of the graph calculation once the distributed calculation is finished is illustrated. In one implementation, instead of displaying the entire 7.2 million nodes in this computation graph (e.g., which may be slow and confusing), a portion of the nodes corresponding to the final results is displayed, and those nodes whose precedents cannot be displayed fully are shown in gray colors (e.g., colors, shapes, border style, and/or the like may be used to distinguish the nodes whose precedents are fully displayed vs those with missing precedents). When a user double clicks any of the nodes, the graph display may expand and show the precedents associated with the selected node. Navigation links 1605 and 1610 facilitate navigating large graphs. Navigation links 1605 allow the user to navigate to precedence nodes and dependency nodes. Navigation link 1610 allows the user to go back to the top-level graph after the user drills down to a packaged atom's inner graph. Navigation link 1610 may show the current location of the graph to help the user navigate.

[00151] FIGURE 17 shows a screen shot diagram illustrating user interface features in one embodiment of the DSGRCE. In Figure 17, an exemplary user interface (e.g., for a mobile device, for a website) showing a logical dependency graph 1701 that provides details regarding a single trade's preceding nodes (e.g., which are only a small portion of the 7.2 million nodes) is illustrated. Different colors may indicate each worker's identity, showing this trade valuation involves three different workers' collaboration (e.g., colors, patterns, shapes, border style,

1 and/or the like may be used to identity which worker is associated with a node).
2 When a user clicks one of the nodes, its values are displayed in panel 1705,
3 allowing the user to inspect any intermediate results in the overall calculation.

4 **[00152]** FIGURE 18 shows a screen shot diagram illustrating user interface
5 features in one embodiment of the DSGRCE. In Figure 18, an exemplary user
6 interface (e.g., for a mobile device, for a website) showing a rule dependency graph
7 1801 corresponding to the logical dependency graph 1601 is illustrated. The rule
8 dependency graph has 40 underlying rules for the 7.2 million calculation nodes.
9 This example shows that it is easier to trace the logic through the 40 nodes of
10 rules, than trying to do the same through the 7.2 million calculation nodes, in
11 order to understand the governing logic of the calculation.

12 **[00153]** FIGURE 19 shows a screen shot diagram illustrating user interface
13 features in one embodiment of the DSGRCE. In Figure 19, an exemplary user
14 interface (e.g., for a mobile device, for a website) for showing sensitivities is
15 illustrated. Sensitivities 1901 associated with the logical dependency graph 1601
16 (e.g., using algorithmic differentiation to obtain results) are shown.

17 **[00154]** FIGURE 20 shows a screen shot diagram illustrating user interface
18 features in one embodiment of the DSGRCE. In Figure 20, an exemplary user
19 interface (e.g., for a mobile device, for a website) for providing debugging
20 capabilities is illustrated. Using the navigation features, a developer may find and
21 inspect any intermediate calculation (e.g., on a remote system running on AWS
22 cloud). When the developer has identified a node whose calculation might be

incorrect, the developer may want to run the same calculation in a debugger and inspect the calculation step by step. However, it may not be feasible to do so on the remote system because the developer might not have the privilege to login and/or the remote system may be a production cloud that does not have any debugging information (e.g., the source or symbol files).

[00155] Instead, in some implementations, the DSGRCE may offer alternative debugging capabilities to debug a remote distributed system without the login privilege or when the production server is run without any debug symbols and information. When the developer wants to debug a given node, the developer may click on the debug link 2001, and a display (e.g., pop-up window) 2005 may show code that the developer may copy/paste to the developer's local machine and execute. After executing this code, the user may get an exact copy of the **Atom** in the local environment, where the developer may use a debugger locally to inspect every step of the calculation in that **Atom** (e.g., the source or symbol files utilized by the debugger should exist on the developer's local computer).

DETAILED DESCRIPTION OF THE DSGRCE COORDINATOR

[00156] FIGURE 21 shows a block diagram illustrating an exemplary DSGRCE coordinator in one embodiment of the DSGRCE. The DSGRCE coordinator facilitates the operation of the DSGRCE via a computer system (e.g., one or more cloud computing systems (e.g., Microsoft Azure, Amazon Web Services, Google Cloud Platform), grid computing systems, virtualized computer systems,

1 mainframe computers, servers, clients, nodes, desktops, mobile devices (e.g., such
2 as smart phones, cellular phones, tablets, personal digital assistants (PDAs),
3 smart watches, and/or the like), embedded computers, dedicated computers,
4 system on a chip (SOC) devices, and/or the like). For example, the DSGRCE
5 coordinator may receive, obtain, aggregate, process, generate, store, retrieve,
6 send, delete, input, output, and/or the like data (including program data and
7 program instructions); may execute program instructions; may communicate with
8 computer systems, with nodes, with users, and/or the like. In various
9 embodiments, the DSGRCE coordinator may comprise a standalone computer
10 system, a distributed computer system, a node in a computer network (i.e., a
11 network of computer systems organized in a topology), a network of DSGRCE
12 coordinators, and/or the like. It is to be understood that the DSGRCE coordinator
13 and/or the various DSGRCE coordinator elements (e.g., processor, system bus,
14 memory, input/output devices) may be organized in any number of ways (i.e.,
15 using any number and configuration of computer systems, computer networks,
16 nodes, DSGRCE coordinator elements, and/or the like) to facilitate DSGRCE
17 operation. Furthermore, it is to be understood that the various DSGRCE
18 coordinator computer systems, DSGRCE coordinator computer networks,
19 DSGRCE coordinator nodes, DSGRCE coordinator elements, and/or the like may
20 communicate among each other in any number of ways to facilitate DSGRCE
21 operation. As used in this disclosure, the term “user” refers generally to people
22 and/or computer systems that interact with the DSGRCE; the term “server” refers

1 generally to a computer system, a program, and/or a combination thereof that
2 handles requests and/or responds to requests from clients via a computer network;
3 the term “client” refers generally to a computer system, a program, a user, and/or
4 a combination thereof that generates requests and/or handles responses from
5 servers via a computer network; the term “node” refers generally to a server, to a
6 client, and/or to an intermediary computer system, program, and/or a combination
7 thereof that facilitates transmission of and/or handling of requests and/or
8 responses.

9 **[00157]** The DSGRCE coordinator includes a processor 2101 that executes
10 program instructions (e.g., DSGRCE program instructions). In various
11 embodiments, the processor may be a general purpose microprocessor (e.g., a
12 central processing unit (CPU), an accelerated processing unit (APU)), a dedicated
13 microprocessor (e.g., a graphics processing unit (GPU), a physics processing unit
14 (PPU), a digital signal processor (DSP), a network processor, a tensor processing
15 unit (TPU), a cryptographic processor, a biometrics processor, and/or the like), an
16 external processor, a plurality of processors (e.g., working in parallel, distributed,
17 and/or the like), a microcontroller (e.g., for an embedded system), and/or the like.
18 The processor may be implemented using integrated circuits (ICs), application-
19 specific integrated circuits (ASICs), field-programmable gate arrays (FPGAs),
20 and/or the like. In various implementations, the processor may comprise one or
21 more cores, may include embedded elements (e.g., a coprocessor such as a math
22 coprocessor, an embedded dedicated microprocessor utilized as a coprocessor,

1 registers, cache memory, software), may be synchronous (e.g., using a clock signal)
2 or asynchronous (e.g., without a central clock), and/or the like. For example, the
3 processor may be an AMD Ryzen processor, an AMD Epyc processor, an Intel Core
4 processor, an Intel Xeon processor, an ARM Cortex processor, an Apple A
5 processor, an Apple S processor, a Qualcomm Snapdragon processor, an IBM
6 PowerPC processor, and/or the like.

7 **[00158]** The processor may be connected to system memory 2105 via a system
8 bus 2103. The system bus may interconnect these and/or other elements of the
9 DSGRCE coordinator via electrical, electronic, optical, wireless, and/or the like
10 communication links (e.g., the system bus may be integrated into a motherboard
11 that interconnects DSGRCE coordinator elements and provides power from a
12 power supply). In various embodiments, the system bus may comprise one or more
13 control buses, address buses, data buses, memory buses, peripheral buses, and/or
14 the like. In various implementations, the system bus may be a parallel bus, a
15 serial bus, a daisy chain design, a hub design, and/or the like. For example, the
16 system bus may comprise a front-side bus, a back-side bus, AMD's Infinity Fabric,
17 Intel's QuickPath Interconnect (QPI), a peripheral component interconnect (PCI)
18 bus, a PCI Express bus, a low pin count (LPC) bus, a universal serial bus (USB),
19 and/or the like. The system memory, in various embodiments, may comprise
20 registers, cache memory (e.g., level one, level two, level three), volatile memory
21 (e.g., random access memory (RAM) (e.g., static RAM (SRAM), dynamic RAM
22 (DRAM)), non-volatile memory (e.g., read only memory (ROM), non-volatile

1 random-access memory (NVRAM) (e.g., resistive random access memory
2 (ReRAM), magnetoresistive random access memory (MRAM)), flash memory (e.g.,
3 NAND-type)), and/or the like. The system memory may utilize error-correcting
4 code (ECC) technology to detect and/or correct internal data corruption. The
5 system memory may be discreet, external, embedded, integrated into a CPU,
6 and/or the like. The processor may access, read from, write to, store in, erase,
7 modify, and/or the like, the system memory in accordance with program
8 instructions (e.g., DSGRCE program instructions) executed by the processor. The
9 system memory may facilitate accessing, storing, retrieving, modifying, deleting,
10 and/or the like data (e.g., DSGRCE data) by the processor.

11 **[00159]** In various embodiments, input/output devices 2110 may be connected to
12 the processor and/or to the system memory, and/or to one another via the system
13 bus.

14 **[00160]** In some embodiments, the input/output devices may include one or more
15 graphics devices 2111. The processor may make use of the one or more graphic
16 devices in accordance with program instructions (e.g., DSGRCE program
17 instructions) executed by the processor. In one implementation, a graphics device
18 may be a video card that may obtain (e.g., via a connected video camera), process
19 (e.g., render a frame), output (e.g., via a connected monitor, television, and/or the
20 like), and/or the like graphical (e.g., multimedia, video, image, text) data (e.g.,
21 DSGRCE data). A video card may be connected to the system bus via an interface
22 such as PCI, PCI Express, USB, PC Card, ExpressCard, Thunderbolt, NVLink,

1 and/or the like. A video card may use one or more graphics processing units
2 (GPUs), for example, by utilizing AMD's CrossFireX and/or NVIDIA's SLI
3 technologies. A video card may be connected via an interface (e.g., video graphics
4 array (VGA), digital video interface (DVI), Mini-DVI, Micro-DVI, high-definition
5 multimedia interface (HDMI), DisplayPort, Thunderbolt, composite video, S-
6 Video, component video, and/or the like) to one or more displays (e.g., cathode ray
7 tube (CRT), liquid crystal display (LCD), touchscreen, video projector, and/or the
8 like) that display graphics. For example, a video card may be an AMD Radeon, an
9 NVIDIA GeForce RTX, an Intel UHD Graphics, and/or the like. In another
10 implementation, a graphics device may be a video capture board that may obtain
11 (e.g., via coaxial cable), process (e.g., overlay with other graphical data), capture,
12 convert (e.g., between different formats, such as MPEG2 to H.264), and/or the like
13 graphical data. A video capture board may be and/or include a TV tuner, may be
14 compatible with a variety of broadcast signals (e.g., NTSC, PAL, ATSC, QAM)
15 may be a part of a video card, and/or the like. For example, a video capture board
16 may be a Hauppauge ImpactVCB, a Hauppauge WinTV-HVR, a Hauppauge
17 Colossus, and/or the like. A graphics device may be discreet, external, embedded,
18 integrated into a CPU, and/or the like. A graphics device may operate in
19 combination with other graphics devices (e.g., in parallel) to provide improved
20 capabilities, data throughput, color depth, and/or the like.

21 **[00161]** In some embodiments, the input/output devices may include one or more
22 audio devices 2113. The processor may make use of the one or more audio devices

1 in accordance with program instructions (e.g., DSGRCE program instructions)
2 executed by the processor. In one implementation, an audio device may be a sound
3 card that may obtain (e.g., via a connected microphone), process, output (e.g., via
4 connected speakers), and/or the like audio data (e.g., DSGRCE data). A sound card
5 may be connected to the system bus via an interface such as PCI, PCI Express,
6 USB, PC Card, ExpressCard, Thunderbolt, and/or the like. A sound card may be
7 connected via an interface (e.g., tip sleeve (TS), tip ring sleeve (TRS), RCA,
8 TOSLINK, optical) to one or more amplifiers, speakers (e.g., mono, stereo,
9 surround sound), subwoofers, digital musical instruments, and/or the like. For
10 example, a sound card may be an Intel AC'97 integrated codec chip, an Intel HD
11 Audio integrated codec chip, a Creative Sound Blaster, and/or the like. An audio
12 device may be discreet, external, embedded, integrated into a motherboard (e.g.,
13 via a chipset), and/or the like. An audio device may operate in combination with
14 other audio devices (e.g., in parallel) to provide improved capabilities, data
15 throughput, audio quality, and/or the like.

16 **[00162]** In some embodiments, the input/output devices may include one or more
17 network devices 2115. The processor may make use of the one or more network
18 devices in accordance with program instructions (e.g., DSGRCE program
19 instructions) executed by the processor. In one implementation, a network device
20 may be a network card that may obtain (e.g., via a Category 6 Ethernet cable),
21 process, output (e.g., via a wireless antenna), and/or the like network data (e.g.,
22 DSGRCE data). A network card may be connected to the system bus via an

1 interface such as PCI, PCI Express, USB, FireWire, PC Card, ExpressCard,
2 Thunderbolt, and/or the like. A network card may be a wired network card (e.g.,
3 10/100/1000BASE-T, optical fiber), a wireless network card (e.g., Wi-Fi
4 802.11ac/ad/ax/ay, Bluetooth, Near Field Communication (NFC), TransferJet), a
5 modem (e.g., dialup telephone-based, asymmetric digital subscriber line (ADSL),
6 cable modem, power line modem, wireless modem based on cellular protocols such
7 as high speed packet access (HSPA), evolution-data optimized (EV-DO), global
8 system for mobile communications (GSM), worldwide interoperability for
9 microwave access (WiMax), long term evolution (LTE), 5G, and/or the like,
10 satellite modem, FM radio modem, radio-frequency identification (RFID) modem,
11 infrared (IR) modem), and/or the like. For example, a network card may be an
12 Intel Gigabit Adapter, a LINKSYS USB Ethernet Adapter, an ASUS wireless
13 Bluetooth and Gigabit WiFi adapter, a Motorola SURFboard Cable Modem, a
14 U.S. Robotics Faxmodem, a Zoom ADSL Modem/Router, a TRENDnet Powerline
15 Ethernet Adapter, a StarTech Gigabit Ethernet Multi Mode Fiber Media
16 Converter, a Broadcom NFC controller, a Qualcomm Snapdragon 4G LTE and 5G
17 modem, a Toshiba TransferJet device, and/or the like. A network device may be
18 discreet, external, embedded, integrated into a motherboard, and/or the like. A
19 network device may operate in combination with other network devices (e.g., in
20 parallel) to provide improved data throughput, redundancy, and/or the like. For
21 example, protocols such as link aggregation control protocol (LACP) (e.g., based
22 on IEEE 802.3AD or IEEE 802.1AX standards) may be used. A network device

1 may be used to connect to a local area network (LAN), a wide area network (WAN),
2 a metropolitan area network (MAN), a personal area network, the Internet, an
3 intranet, a Bluetooth network, an NFC network, a Wi-Fi network, a cellular
4 network, and/or the like.

5 **[00163]** In some embodiments, the input/output devices may include one or more
6 peripheral devices 2117. The processor may make use of the one or more
7 peripheral devices in accordance with program instructions (e.g., DSGRCE
8 program instructions) executed by the processor. In various implementations, a
9 peripheral device may be a digital camera, a video camera, a webcam, an
10 electronically moveable pan tilt zoom (PTZ) camera, a monitor, a touchscreen
11 display, active shutter 3D glasses, head-tracking 3D glasses, a virtual reality (VR)
12 headset, an augmented reality (AR) headset, a remote control, an audio line-in, an
13 audio line-out, a microphone, headphones, speakers, a subwoofer, an amplifier, a
14 router, a hub, a switch, a firewall, an antenna, a keyboard, a mouse, a trackpad,
15 a trackball, a digitizing tablet, a stylus, a joystick, a gamepad, a game controller,
16 a force-feedback device, a laser, sensors (e.g., proximity sensor, rangefinder,
17 ambient temperature sensor, ambient light sensor, humidity sensor, an
18 accelerometer, a gyroscope, a motion sensor, an olfaction sensor, a biosensor, a
19 biometric sensor, a chemical sensor, a magnetometer, a radar, a sonar, a location
20 sensor such as global positioning system (GPS), Galileo, GLONASS, and/or the
21 like), a printer, a fax, a scanner, a copier, a card reader, a fingerprint reader, a pin
22 entry device (PED), a Trusted Platform Module (TPM), a hardware security

1 module (HSM), and/or the like. A peripheral device may be connected to the
2 system bus via an interface such as PCI, PCI Express, USB, FireWire, VGA, DVI,
3 Mini-DVI, Micro-DVI, HDMI, DisplayPort, Thunderbolt, composite video, S-
4 Video, component video, PC Card, ExpressCard, serial port, parallel port, PS/2,
5 TS, TRS, RCA, TOSLINK, network connection (e.g., wired such as Ethernet,
6 optical fiber, and/or the like; wireless such as Wi-Fi, Bluetooth, NFC, cellular, 5G,
7 and/or the like), a connector of another input/output device, and/or the like. A
8 peripheral device may be discreet, external, embedded, integrated (e.g., into a
9 processor, into a motherboard), and/or the like. A peripheral device may operate
10 in combination with other peripheral devices (e.g., in parallel) to provide the
11 DSGRCE coordinator with a variety of input, output and processing capabilities.

12 **[00164]** In some embodiments, the input/output devices may include one or more
13 storage devices 2119. The processor may access, read from, write to, store in,
14 erase, modify, and/or the like a storage device in accordance with program
15 instructions (e.g., DSGRCE program instructions) executed by the processor. A
16 storage device may facilitate accessing, storing, retrieving, modifying, deleting,
17 and/or the like data (e.g., DSGRCE data) by the processor. In one implementation,
18 the processor may access data from the storage device directly via the system bus.
19 In another implementation, the processor may access data from the storage device
20 by instructing the storage device to transfer the data to the system memory and
21 accessing the data from the system memory. In various embodiments, a storage
22 device may be a hard disk drive (HDD), a solid-state drive (SSD), an optical disk

1 drive (e.g., compact disk (CD-ROM) drive, CD-Recordable (CD-R) drive, CD-
2 Rewriteable (CD-RW) drive, digital versatile disc (DVD-ROM) drive, DVD-R drive,
3 DVD-RW drive, Blu-ray disk (BD) drive) using an optical medium, a magnetic tape
4 drive using a magnetic tape, a memory card (e.g., a USB flash drive, a compact
5 flash (CF) card, a secure digital extended capacity (SDXC) card), a network
6 attached storage (NAS), a direct-attached storage (DAS), a storage area network
7 (SAN), other processor-readable physical mediums, and/or the like. A storage
8 device may be connected to the system bus via an interface such as PCI, PCI
9 Express, USB, FireWire, PC Card, ExpressCard, Thunderbolt, integrated drive
10 electronics (IDE), serial advanced technology attachment (SATA), non-volatile
11 memory express (NVMe), external SATA (eSATA), small computer system
12 interface (SCSI), serial attached SCSI (SAS), fibre channel (FC), network
13 connection (e.g., wired such as Ethernet, optical fiber, and/or the like; wireless
14 such as Wi-Fi, Bluetooth, NFC, cellular, 5G, and/or the like), and/or the like. A
15 storage device may be discreet, external, embedded, integrated (e.g., into a
16 motherboard, into another storage device), and/or the like. A storage device may
17 operate in combination with other storage devices to provide improved capacity,
18 data throughput, data redundancy, and/or the like. For example, protocols such
19 as redundant array of independent disks (RAID) (e.g., RAID 0 (striping), RAID 1
20 (mirroring), RAID 5 (striping with distributed parity), hybrid RAID), just a bunch
21 of drives (JBOD), and/or the like may be used. In another example, virtual and/or
22 physical drives may be pooled to create a storage pool (e.g., for virtual storage,

cloud storage, and/or the like). In yet another example, an SSD cache may be used with an HDD to improve speed.

[00165] Together and/or separately the system memory 2105 and the one or more storage devices 2119 may be referred to as memory 2120 (i.e., physical memory).

[00166] DSGRCE memory 2120 contains processor-operable (e.g., accessible) DSGRCE data stores 2130. Data stores 2130 comprise data that may be used (e.g., by the DSGRCE) via the DSGRCE coordinator. Such data may be organized using one or more data formats such as a database (e.g., a relational database with database tables, an object-oriented database, a graph database, a hierarchical database, a key-value NoSQL database), a flat file (e.g., organized into a tabular format), a binary file (e.g., a GIF file, an MPEG-4 file), a structured file (e.g., an HTML file, an XML file), a text file, and/or the like. Furthermore, data may be organized using one or more data structures such as an array, a queue, a stack, a set, a linked list, a map, a tree, a hash, a record, an object, a directed graph, and/or the like. In various embodiments, data stores may be organized in any number of ways (i.e., using any number and configuration of data formats, data structures, DSGRCE coordinator elements, and/or the like) to facilitate DSGRCE operation. For example, DSGRCE data stores may comprise data stores 2130a-d implemented as one or more databases. A users data store 2130a may be a collection of database tables that include fields such as UserID, UserName, UserPreferences, and/or the like. A clients data store 2130b may be a collection of

1 database tables that include fields such as ClientID, ClientName,
2 ClientDeviceType, ClientScreenResolution, and/or the like. An atoms data store
3 2130c may be a collection of database tables that include fields such as AtomID,
4 AtomName, AtomParameters, AtomVersion, AtomDeveloperID,
5 AtomAccessPriveleges, and/or the like. A rules data store 2130d may be a
6 collection of database tables that include fields such as RuleID, RuleNamespace,
7 RuleName, RuleParameters, RulePrecedentRules, RuleDependentRules,
8 RuleAtomID, RuleAtomParameters, RuleAccessPriveleges, and/or the like. The
9 DSGRCE coordinator may use data stores 2130 to keep track of inputs,
10 parameters, settings, variables, records, outputs, and/or the like.

11 **[00167]** DSGRCE memory 2120 contains processor- operable (e.g., executable)
12 DSGRCE components 2140. Components 2140 comprise program components
13 (including program instructions and any associated data stores) that are executed
14 (e.g., by the DSGRCE) via the DSGRCE coordinator (i.e., via the processor) to
15 transform DSGRCE inputs into DSGRCE outputs. It is to be understood that the
16 various components and their subcomponents, capabilities, applications, and/or
17 the like may be organized in any number of ways (i.e., using any number and
18 configuration of components, subcomponents, capabilities, applications, DSGRCE
19 coordinator elements, and/or the like) to facilitate DSGRCE operation.
20 Furthermore, it is to be understood that the various components and their
21 subcomponents, capabilities, applications, and/or the like may communicate
22 among each other in any number of ways to facilitate DSGRCE operation. For

1 example, the various components and their subcomponents, capabilities,
2 applications, and/or the like may be combined, integrated, consolidated, split up,
3 distributed, and/or the like in any number of ways to facilitate DSGRCE
4 operation. In another example, a single or multiple instances of the various
5 components and their subcomponents, capabilities, applications, and/or the like
6 may be instantiated on each of a single DSGRCE coordinator node, across multiple
7 DSGRCE coordinator nodes, and/or the like.

8 **[00168]** In various embodiments, program components may be developed using
9 one or more programming languages, techniques, tools, and/or the like such as an
10 assembly language, Ada, BASIC, C, C++, C#, COBOL, Fortran, Java, LabVIEW,
11 Lisp, Mathematica, MATLAB, OCaml, PL/I, Smalltalk, Visual Basic for
12 Applications (VBA), HTML, XML, CSS, JavaScript, JavaScript Object Notation
13 (JSON), PHP, Perl, Ruby, Python, Asynchronous JavaScript and XML (AJAX),
14 WebSocket Protocol, Simple Object Access Protocol (SOAP), SSL, ColdFusion,
15 Microsoft .NET, Apache modules, Adobe Flash, Adobe AIR, Microsoft Silverlight,
16 Windows PowerShell, batch files, Tcl, graphical user interface (GUI) toolkits,
17 SQL, database adapters, web application programming interfaces (APIs), web
18 frameworks (e.g., Ruby on Rails, AngularJS), application server extensions,
19 integrated development environments (IDEs), libraries (e.g., object libraries, class
20 libraries, remote libraries), remote procedure calls (RPCs), Common Object
21 Request Broker Architecture (CORBA), and/or the like.

1 **[00169]** In some embodiments, components 2140 may include an operating
2 environment component 2140a. The operating environment component may
3 facilitate operation of the DSGRCE via various subcomponents.

4 **[00170]** In some implementations, the operating environment component may
5 include an operating system subcomponent. The operating system subcomponent
6 may provide an abstraction layer that facilitates the use of, communication
7 among, common services for, interaction with, security of, and/or the like of
8 various DSGRCE coordinator elements, components, data stores, and/or the like.

9 **[00171]** In some embodiments, the operating system subcomponent may
10 facilitate execution of program instructions (e.g., DSGRCE program instructions)
11 by the processor by providing process management capabilities. For example, the
12 operating system subcomponent may facilitate the use of multiple processors, the
13 execution of multiple processes, the execution of multiple threads, multitasking,
14 and/or the like.

15 **[00172]** In some embodiments, the operating system subcomponent may
16 facilitate the use of memory by the DSGRCE. For example, the operating system
17 subcomponent may allocate and/or free memory, facilitate memory addressing,
18 provide memory segmentation and/or protection, provide virtual memory
19 capability, facilitate caching, and/or the like. In another example, the operating
20 system subcomponent may include a file system (e.g., File Allocation Table (FAT),
21 New Technology File System (NTFS), Hierarchical File System Plus (HFS+),
22 Apple File System (APFS), Universal Disk Format (UDF), Linear Tape File

1 System (LTFS)) to facilitate storage, retrieval, deletion, aggregation, processing,
2 generation, and/or the like of data.

3 **[00173]** In some embodiments, the operating system subcomponent may
4 facilitate operation of and/or processing of data for and/or from input/output
5 devices. For example, the operating system subcomponent may include one or
6 more device drivers, interrupt handlers, file systems, and/or the like that allow
7 interaction with input/output devices.

8 **[00174]** In some embodiments, the operating system subcomponent may
9 facilitate operation of the DSGRCE coordinator as a node in a computer network
10 by providing support for one or more communications protocols. For example, the
11 operating system subcomponent may include support for the internet protocol
12 suite (i.e., Transmission Control Protocol/Internet Protocol (TCP/IP)) of network
13 protocols such as IP, IPsec, Mobile IP, TCP, User Datagram Protocol (UDP),
14 and/or the like. In another example, the operating system subcomponent may
15 include support for security protocols (e.g., Wired Equivalent Privacy (WEP), Wi-
16 Fi Protected Access (WPA), WPA2, WPA3) for wireless computer networks. In yet
17 another example, the operating system subcomponent may include support for
18 virtual private networks (VPNs).

19 **[00175]** In some embodiments, the operating system subcomponent may
20 facilitate security of the DSGRCE coordinator. For example, the operating system
21 subcomponent may provide services such as authentication, authorization, audit,

1 network intrusion-detection capabilities, firewall capabilities, antivirus
2 capabilities, and/or the like.

3 **[00176]** In some embodiments, the operating system subcomponent may
4 facilitate user interaction with the DSGRCE by providing user interface elements
5 that may be used by the DSGRCE to generate a user interface. In one
6 implementation, such user interface elements may include widgets (e.g., windows,
7 dialog boxes, scrollbars, menu bars, tabs, ribbons, menus, buttons, text boxes,
8 checkboxes, combo boxes, drop-down lists, list boxes, radio buttons, sliders,
9 spinners, grids, labels, progress indicators, icons, tooltips, and/or the like) that
10 may be used to obtain input from and/or provide output to the user. For example,
11 such widgets may be used via a widget toolkit such as Microsoft Foundation
12 Classes (MFC), Apple Cocoa Touch, Java Swing, JavaFX, jQuery UI, GTK, Qt,
13 and/or the like. In another implementation, such user interface elements may
14 include sounds (e.g., event notification sounds stored in MP3 file format),
15 animations, vibrations, and/or the like that may be used to inform the user
16 regarding occurrence of various events. For example, the operating system
17 subcomponent may include a user interface such as Windows Aero, Windows
18 Metro, macOS X Aqua, macOS X Flat, GNOME Shell, KDE Plasma Workspaces
19 (e.g., Plasma Desktop, Plasma Netbook, Plasma Contour, Plasma Mobile), and/or
20 the like.

21 **[00177]** In various embodiments the operating system subcomponent may
22 comprise a single-user operating system, a multi-user operating system, a single-

1 tasking operating system, a multitasking operating system, a single-processor
2 operating system, a multiprocessor operating system, a distributed operating
3 system, an embedded operating system, a real-time operating system, and/or the
4 like. For example, the operating system subcomponent may comprise an operating
5 system such as UNIX, LINUX, IBM i, Sun Solaris, Microsoft Windows Server,
6 Microsoft DOS, Microsoft Windows 10, Apple Mac OS X, Apple iOS, Google
7 Android, and/or the like.

8 **[00178]** In some implementations, the operating environment component may
9 include a database subcomponent. The database subcomponent may facilitate
10 DSGRCE capabilities such as storage, analysis, retrieval, access, modification,
11 deletion, aggregation, generation, and/or the like of data (e.g., the use of data
12 stores 2130). The database subcomponent may make use of database languages
13 (e.g., Structured Query Language (SQL), XQuery, Cassandra Query Language
14 (CQL)), stored procedures, triggers, APIs, and/or the like to provide these
15 capabilities. In various embodiments the database subcomponent may comprise a
16 cloud database, a data warehouse, a distributed database, an embedded database,
17 a parallel database, a real-time database, and/or the like. For example, the
18 database subcomponent may comprise a database such as Microsoft SQL Server,
19 Microsoft Access, MySQL, IBM DB2, Oracle Database, Apache Cassandra
20 database, MongoDB, and/or the like.

21 **[00179]** In some implementations, the operating environment component may
22 include an information handling subcomponent. The information handling

1 subcomponent may provide the DSGRCE with capabilities to serve, deliver,
2 upload, obtain, present, download, and/or the like a variety of information. The
3 information handling subcomponent may use protocols such as Hypertext
4 Transfer Protocol (HTTP), Hypertext Transfer Protocol Secure (HTTPS), File
5 Transfer Protocol (FTP), Telnet, Secure Shell (SSH), Transport Layer Security
6 (TLS), Secure Sockets Layer (SSL), peer-to-peer (P2P) protocols (e.g., BitTorrent,
7 Bitcoin), and/or the like to handle communication of information such as web
8 pages, files, multimedia content (e.g., streaming media), applications,
9 transactions, and/or the like.

10 **[00180]** In some embodiments, the information handling subcomponent may
11 facilitate the serving of information to users, DSGRCE components, nodes in a
12 computer network, web browsers, and/or the like. For example, the information
13 handling subcomponent may comprise a web server such as Apache HTTP Server,
14 Microsoft Internet Information Services (IIS), Oracle WebLogic Server, Adobe
15 Flash Media Server, Adobe Content Server, and/or the like. Furthermore, a web
16 server may include extensions, plug-ins, add-ons, servlets, and/or the like. For
17 example, these may include Apache modules, IIS extensions, Java servlets, and/or
18 the like. In some implementations, the information handling subcomponent may
19 communicate with the database subcomponent via standards such as Open
20 Database Connectivity (ODBC), Java Database Connectivity (JDBC), ActiveX
21 Data Objects for .NET (ADO.NET), and/or the like. For example, the information
22 handling subcomponent may use such standards to store, analyze, retrieve,

1 access, modify, delete, aggregate, generate, and/or the like data (e.g., data from
2 data stores 2130) via the database subcomponent.

3 **[00181]** In some embodiments, the information handling subcomponent may
4 facilitate presentation of information obtained from users, DSGRCE components,
5 nodes in a computer network, web servers, and/or the like. For example, the
6 information handling subcomponent may comprise a web browser such as
7 Microsoft Internet Explorer, Microsoft Edge, Mozilla Firefox, Apple Safari, Google
8 Chrome, Opera Mobile, Amazon Silk, Nintendo 3DS Internet Browser, and/or the
9 like. Furthermore, a web browser may include extensions, plug-ins, add-ons,
10 applets, and/or the like. For example, these may include Adobe Flash Player,
11 Adobe Acrobat plug-in, Microsoft Silverlight plug-in, Microsoft Office plug-in,
12 Java plug-in, and/or the like.

13 **[00182]** In some implementations, the operating environment component may
14 include a messaging subcomponent. The messaging subcomponent may facilitate
15 DSGRCE message communications capabilities. The messaging subcomponent
16 may use protocols such as Simple Mail Transfer Protocol (SMTP), Internet
17 Message Access Protocol (IMAP), Post Office Protocol (POP), Extensible
18 Messaging and Presence Protocol (XMPP), Real-time Transport Protocol (RTP),
19 Internet Relay Chat (IRC), Skype protocol, Messaging Application Programming
20 Interface (MAPI), Facebook API, a custom protocol, and/or the like to facilitate
21 DSGRCE message communications. The messaging subcomponent may facilitate
22 message communications such as email, instant messaging, Voice over IP (VoIP),

1 video conferencing, Short Message Service (SMS), web chat, in-app messaging
2 (e.g., alerts, notifications), and/or the like. For example, the messaging
3 subcomponent may comprise Microsoft Exchange Server, Microsoft Outlook,
4 Sendmail, IBM Domino, IBM Notes, Gmail, ICQ, Trillian, Skype, Google
5 Hangouts, Apple FaceTime, Apple iChat, Facebook Chat, and/or the like.

6 **[00183]** In some implementations, the operating environment component may
7 include a security subcomponent that facilitates DSGRCE security. In some
8 embodiments, the security subcomponent may restrict access to the DSGRCE, to
9 one or more services provided by the DSGRCE, to data associated with the
10 DSGRCE (e.g., stored in data stores 2130), to communication messages associated
11 with the DSGRCE, and/or the like to authorized users. Access may be granted via
12 a login screen, via an API that obtains authentication information, via an
13 authentication token, via a cryptographic key (e.g., stored in an HSM), and/or the
14 like. For example, the user may obtain access by providing a username and/or a
15 password (e.g., a string of characters, a picture password), a personal
16 identification number (PIN), an identification card, a magnetic stripe card, a
17 smart card, a biometric identifier (e.g., a finger print, a voice print, a retina scan,
18 a face scan), a gesture (e.g., a swipe), a media access control (MAC) address, an IP
19 address, and/or the like. Various security models such as access-control lists
20 (ACLs), capability-based security, hierarchical protection domains, multi-factor
21 authentication, and/or the like may be used to control access. For example, the
22 security subcomponent may facilitate digital rights management (DRM), network

1 intrusion detection, firewall capabilities, cryptographic wallet access (e.g., for
2 cryptocurrencies such as Bitcoin, Ethereum, and/or the like), and/or the like.

3 **[00184]** In some embodiments, the security subcomponent may use
4 cryptographic techniques to secure information (e.g., by storing encrypted data),
5 verify message authentication (e.g., via a digital signature), provide integrity
6 checking (e.g., a checksum), authorize transactions (e.g., using a private key),
7 and/or the like by facilitating encryption and/or decryption of data. Furthermore,
8 steganographic techniques may be used instead of or in combination with
9 cryptographic techniques. Cryptographic techniques used by the DSGRCE may
10 include symmetric key cryptography using shared keys (e.g., using one or more
11 block ciphers such as triple Data Encryption Standard (DES), Advanced
12 Encryption Standard (AES); stream ciphers such as Rivest Cipher 4 (RC4),
13 Rabbit), asymmetric key cryptography using a public key/private key pair (e.g.,
14 using algorithms such as Rivest-Shamir-Adleman (RSA), Elliptic Curve Digital
15 Signature Algorithm (ECDSA)), cryptographic hash functions (e.g., using
16 algorithms such as Message-Digest 5 (MD5), Secure Hash Algorithm 3 (SHA-3)),
17 and/or the like. For example, the security subcomponent may comprise a
18 cryptographic system such as Pretty Good Privacy (PGP).

19 **[00185]** In some implementations, the operating environment component may
20 include a virtualization subcomponent that facilitates DSGRCE virtualization
21 capabilities. The virtualization subcomponent may include hypervisors (e.g.,
22 Type-1 native hypervisors, Type-2 hosted hypervisors), virtual machines (VMs),

1 and/or the like. In some embodiments, the virtualization subcomponent may
2 provide support for platform virtualization (e.g., via a virtual machine). Platform
3 virtualization types may include full virtualization, partial virtualization,
4 paravirtualization, and/or the like. In some implementations, platform
5 virtualization may be hardware-assisted (e.g., via support from the processor
6 using technologies such as AMD-V, Intel VT-x, and/or the like). In some
7 embodiments, the virtualization subcomponent may provide support for various
8 other virtualized environments such as via operating-system level virtualization,
9 desktop virtualization, workspace virtualization, mobile virtualization,
10 application virtualization, database virtualization, and/or the like. In some
11 embodiments, the virtualization subcomponent may provide support for various
12 virtualized resources such as via processing resources virtualization, memory
13 virtualization, storage virtualization, data virtualization, network virtualization,
14 and/or the like. For example, the virtualization subcomponent may comprise
15 VMware software suite (e.g., VMware Workstation, VMware Player, VMware
16 ESXi, VMware ThinApp, VMware Infrastructure), Parallels software suite (e.g.,
17 Parallels Server, Parallels Desktop), Virtuozzo software suite (e.g., Virtuozzo
18 Infrastructure Platform, Virtuozzo), Oracle software suite (e.g., Oracle VM Server
19 for SPARC, Oracle VM Server for x86, Oracle VM VirtualBox, Oracle Solaris 11),
20 Wine, and/or the like.

21 **[00186]** In some embodiments, components 2140 may include a user interface
22 component 2140b. The user interface component may facilitate user interaction

1 with the DSGRCE by providing a user interface. In various implementations, the
2 user interface component may include programmatic instructions to obtain input
3 from and/or provide output to the user via physical controls (e.g., physical buttons,
4 switches, knobs, wheels, dials), textual user interface, audio user interface, GUI,
5 voice recognition, gesture recognition, touch and/or multi-touch user interface,
6 messages, APIs, and/or the like. In some implementations, the user interface
7 component may make use of the user interface elements provided by the operating
8 system subcomponent of the operating environment component. For example, the
9 user interface component may make use of the operating system subcomponent's
10 user interface elements via a widget toolkit. In some implementations, the user
11 interface component may make use of information presentation capabilities
12 provided by the information handling subcomponent of the operating environment
13 component. For example, the user interface component may make use of a web
14 browser to provide a user interface via HTML5, Adobe Flash, Microsoft
15 Silverlight, and/or the like.

16 **[00187]** In some embodiments, components 2140 may include any of the
17 components RP 2140c, LDGG 2140d, WSD 2140e, WI 2140f, GE 2140g, WC 2140h,
18 etc. described in more detail in preceding figures.

19 **[00188]** Additional embodiments may include:

- 20 1. A distributed system generating rule compiler apparatus, comprising:
21 a memory;

1 a processor in communication with the memory, and configured to issue a
2 plurality of processing instructions stored in the memory, wherein the processor
3 issues instructions to:

4 (a) obtain, via the processor, a distributed system execution request data
5 structure;

6 (b) determine, via the processor, an output rule and the output rule's set
7 of rule parameters specified via the distributed system execution request data
8 structure for a requested calculation, wherein a current rule is initialized to the
9 output rule;

10 (c) query, via the processor, a rule lookup table data structure to
11 determine a set of matching rules, corresponding to the current rule, based on the
12 current rule's identification tuple;

13 (d) select, via the processor, the best matching rule from the set of
14 matching rules, wherein the best matching rule's set of rule parameters best
15 corresponds to the current rule's set of rule parameters;

16 (e) generate, via the processor, a logical dependency graph data
17 structure by adding logical dependency graph nodes and logical dependency graph
18 edges corresponding to the best matching rule, precedent rules of the best matching
19 rule, and precedent rules of each precedent rule;

20 (f) determine, via the processor, an execution complexity gauge value
21 associated with the generated logical dependency graph data structure;

1 (g) determine, via the processor, a set of distributed worker processes to
2 be utilized to execute the requested calculation;

3 (h) divide, via the processor, the generated logical dependency graph
4 data structure into a set of subgraphs based on the execution complexity gauge value,
5 wherein each worker process in the set of distributed worker processes is assigned a
6 subgraph from the set of subgraphs;

7 (i) initialize, via the processor, each respective worker process in the set
8 of distributed worker processes with the subgraph assigned to the respective worker
9 process;

10 (j) coordinate, via the processor, execution of the requested calculation
11 by worker processes in the set of distributed worker processes, wherein each
12 respective worker process calculates results for logical dependency graph nodes in the
13 subgraph assigned to the respective worker process; and

14 (k) obtain, via the processor, a computation result of the logical
15 dependency graph node corresponding to the output rule, from the worker process in
16 the set of distributed worker processes that is assigned the subgraph that contains
17 the logical dependency graph node corresponding to the output rule.

18 2. The apparatus of embodiment 1, wherein the rule lookup table data structure
19 contains a set of identification tuple to rule data structure mappings.

1 3. The apparatus of embodiment 2, wherein a rule data structure includes a data
2 field that references an associated concrete atom data structure that implements a
3 forward calculation operation of a generic atom interface to calculate a result.

4 4. The apparatus of embodiment 3, wherein a rule data structure includes a data
5 field that references a factory function that creates an instance of the associated
6 concrete atom data structure.

7 5. The apparatus of one of the embodiments 1-4, wherein an identification tuple
8 comprises an ordered list that includes a rule's namespace and rule name.

9 6. The apparatus of one of the embodiments 1-4, wherein an identification tuple
10 comprises a hash value calculated based on an ordered list that includes a rule's
11 namespace and rule name.

12 7. The apparatus of one of the embodiments 1-6, wherein a single rule exists in
13 the set of matching rules, and the single rule is selected as the best matching rule.

14 8. The apparatus of one of the embodiments 1-6, wherein multiple rules exist in
15 the set of matching rules, and a rule with the minimum inheritance distance is
16 selected as the best matching rule.

17 9. The apparatus of embodiment 8, wherein multiple rules with the minimum
18 inheritance distance exist, and the best matching rule is selected using a tie breaker
19 technique based on one of: position of rule parameters, types of rule parameters.

1 10. The apparatus of one of the embodiments 1-9, wherein the instructions to (e)
2 generate the logical dependency graph data structure further comprise instructions
3 to:

4 (e1) determine, via the processor, a logical dependency graph node identifier
5 for the best matching rule;

6 (e2) either, upon determining, via the processor, that the logical dependency
7 graph node identifier does not yet exist in the logical dependency graph data
8 structure:

9 add, via the processor, a new logical dependency graph node,
10 corresponding to the best matching rule, to the logical dependency graph data
11 structure, wherein the new logical dependency graph node is assigned the logical
12 dependency graph node identifier;

13 upon determining, via the processor, that the current rule is not the
14 output rule, add a logical dependency graph edge between the new logical dependency
15 graph node and a logical dependency graph node corresponding to the best matching
16 rule's dependent rule;

17 determine, via the processor, a set of precedent rules for the best
18 matching rule, wherein the best matching rule is a dependent rule for precedent rules
19 in the set of precedent rules; and

1 process, via the processor, each respective precedent rule in the set of
2 precedent rules by: setting the current rule to the respective precedent rule, and
3 repeating elements (c)-(e); and

4 return, via the processor, to processing precedent rules of the best
5 matching rule's dependent rule until precedent rules for the output rule are
6 processed;

7 (e3) or, upon determining, via the processor, that the logical dependency graph
8 node identifier already exists in the logical dependency graph data structure:

9 add, via the processor, a logical dependency graph edge between an
10 existing logical dependency graph node, associated with the logical dependency graph
11 node identifier, and a logical dependency graph node corresponding to the best
12 matching rule's dependent rule; and

13 return, via the processor, to processing precedent rules of the best
14 matching rule's dependent rule.

15 11. The apparatus of one of the embodiments 1-10, wherein a logical dependency
16 graph node is a data structure that includes a data field that that references a
17 corresponding rule's identification tuple.

18 12. The apparatus of one of the embodiments 1-11, wherein a logical dependency
19 graph edge is a data field in a logical dependency graph node that refers to another
20 logical dependency graph node.

1 13. The apparatus of one of the embodiments 1-12, wherein the execution
2 complexity gauge value is determined based on at least one of: the number of nodes
3 in the logical dependency graph, the sum of output tensor sizes of logical dependency
4 graph nodes, network data transfer size associated with the logical dependency
5 graph, computation time of the requested calculation.

6 14. The apparatus of one of the embodiments 1-13, wherein the cardinality of the
7 set of distributed worker processes is determined based on a specified fixed number
8 of available worker processes.

9 15. The apparatus of one of the embodiments 1-13, wherein the cardinality of the
10 set of distributed worker processes is determined based on the execution complexity
11 gauge value and a scaling parameter value, wherein the scaling parameter value is
12 determined based on at least one of: target processing time, target available memory,
13 target cost, target data transfer size.

14 16. The apparatus of embodiment 15, wherein the cardinality of the set of
15 distributed worker processes is further determined based on an average worker
16 execution capability.

17 17. The apparatus of one of the embodiments 1-16, wherein the number of nodes
18 in a subgraph assigned to each respective worker process is determined based on the
19 respective worker's worker execution capability, wherein worker execution capability
20 is determined based on at least one of: processor speed, memory size, network speed.

1 18. The apparatus of one of the embodiments 1-17, wherein at least some of logical
2 dependency graph nodes in the generated logical dependency graph data structure
3 are duplicated in multiple subgraphs that are assigned to different worker processes.

4 19. The apparatus of one of the embodiments 1-18, wherein the generated logical
5 dependency graph data structure includes a packaged logical dependency graph node
6 that utilizes an internal logical dependency graph data structure, wherein the
7 internal logical dependency graph data structure is constructed using an associated
8 packaged atom data structure.

9 20. The apparatus of embodiment 19, wherein the internal logical dependency
10 graph data structure is in the same memory space as the worker process in the set of
11 distributed worker processes that is assigned the subgraph that contains the
12 packaged logical dependency graph node.

13 21. The apparatus of embodiment 19, wherein the internal logical dependency
14 graph data structure is distributed across multiple worker processes in the set of
15 distributed worker processes.

16 22. The apparatus of one of the embodiments 1-21, further comprising:

17 the processor issues instructions to:

18 provide a graph representation of the logical dependency graph data
19 structure;

20 obtain a user selection of a target node in the graph representation;

1 determine a target logical dependency graph node in the logical
2 dependency graph data structure corresponding to the user selected target node in
3 the graph representation;

4 obtain a computation result of the target logical dependency graph node
5 from the worker process in the set of distributed worker processes that is assigned
6 the subgraph that contains the target logical dependency graph node; and

7 provide the computation result for display.

8 23. The apparatus of embodiment 22, wherein the target node in the graph
9 representation includes a visual indicator that specifies the worker process in the set
10 of distributed worker processes that is assigned the subgraph that contains the target
11 logical dependency graph node.

12 24. The apparatus of embodiment 23, wherein the visual indicator is at least one
13 of: color, color pattern, shape, or border style of the target node.

14 25. The apparatus of embodiment 22, wherein each node in the graph
15 representation includes a visual indicator that specifies whether the respective node's
16 precedent nodes are displayed.

17 26. The apparatus of embodiment 25, wherein the visual indicator is at least one
18 of: color, color pattern, shape, or border style of the respective node.

19 27. The apparatus of one of the embodiments 22-26, further comprising:
20 the processor issues instructions to:

1 obtain a user command to view state replication code, wherein execution
2 of state replication code on a remote client replicates state of an associated logical
3 dependency graph node on the remote client;

4 determine state replication code associated with the target logical
5 dependency graph node; and

6 generate a display containing the determined state replication code.

7 28. The apparatus of one of the embodiments 22-27, further comprising:

8 the processor issues instructions to:

9 obtain a user command to switch graph display mode;

10 provide a second graph representation of a rule dependency graph data
11 structure corresponding to the logical dependency graph data structure;

12 obtain a user selection of a second target node in the second graph
13 representation; and

14 provide display information associated with the second target node,
15 wherein the display information includes a list of logical dependency graph nodes
16 associated with the second target node and metadata associated with the second
17 target node.

18 29. The apparatus of one of the embodiments 1-21, further comprising:

19 the processor issues instructions to:

1 provide a graph representation of the logical dependency graph data
2 structure;

3 obtain a user selection of a target node in the graph representation;

4 determine a target logical dependency graph node in the logical
5 dependency graph data structure corresponding to the user selected target node in
6 the graph representation; and

7 provide a second graph representation of the target logical dependency
8 graph node's internal logical dependency graph data structure upon determining that
9 the target logical dependency graph node is a packaged atom.

10 30. The apparatus of embodiment 29, further comprising:

11 the processor issues instructions to:

12 obtain a user command to switch graph display mode; and

13 provide a third graph representation of a rule dependency graph data
14 structure corresponding to the target logical dependency graph node's internal logical
15 dependency graph data structure.

16 31. The apparatus of embodiment 30, wherein the third graph representation
17 further includes a graph representation of a rule dependency graph data structure
18 corresponding to the logical dependency graph data structure.

19 32. The apparatus of one of the embodiments 30-31, wherein nested graph
20 representations of rule dependency graph data structures include navigation links
21 that facilitate navigation among the nested graph representations.

1 33. The apparatus of one of the embodiments 1-32, wherein at least one of: the best
2 matching rule, a precedent rule of the best matching rule, and a precedent rule of
3 another precedent rule utilizes dynamic precedents.

4 34. The apparatus of one of the embodiments 1-33, wherein at least one of: the best
5 matching rule, a precedent rule of the best matching rule, and a precedent rule of
6 another precedent rule corresponds to a rule entry in the rule lookup table data
7 structure for a cloned namespace.

8 35. The apparatus of one of the embodiments 1-34, wherein at least one of: the best
9 matching rule, a precedent rule of the best matching rule, and a precedent rule of
10 another precedent rule is an overriding rule in a cloned namespace determined via a
11 namespace lookup table data structure.

12

13

14 101. A distributed system generating rule compiler processor-readable non-
15 transitory physical medium storing processor-issuable instructions to:

16 (a) obtain, via the processor, a distributed system execution request data
17 structure;

18 (b) determine, via the processor, an output rule and the output rule's set of rule
19 parameters specified via the distributed system execution request data structure for
20 a requested calculation, wherein a current rule is initialized to the output rule;

1 (c) query, via the processor, a rule lookup table data structure to determine a
2 set of matching rules, corresponding to the current rule, based on the current rule's
3 identification tuple;

4 (d) select, via the processor, the best matching rule from the set of matching
5 rules, wherein the best matching rule's set of rule parameters best corresponds to the
6 current rule's set of rule parameters;

7 (e) generate, via the processor, a logical dependency graph data structure by
8 adding logical dependency graph nodes and logical dependency graph edges
9 corresponding to the best matching rule, precedent rules of the best matching rule,
10 and precedent rules of each precedent rule;

11 (f) determine, via the processor, an execution complexity gauge value
12 associated with the generated logical dependency graph data structure;

13 (g) determine, via the processor, a set of distributed worker processes to be
14 utilized to execute the requested calculation;

15 (h) divide, via the processor, the generated logical dependency graph data
16 structure into a set of subgraphs based on the execution complexity gauge value,
17 wherein each worker process in the set of distributed worker processes is assigned a
18 subgraph from the set of subgraphs;

19 (i) initialize, via the processor, each respective worker process in the set of
20 distributed worker processes with the subgraph assigned to the respective worker
21 process;

1 (j) coordinate, via the processor, execution of the requested calculation by
2 worker processes in the set of distributed worker processes, wherein each respective
3 worker process calculates results for logical dependency graph nodes in the subgraph
4 assigned to the respective worker process; and

5 (k) obtain, via the processor, a computation result of the logical dependency
6 graph node corresponding to the output rule, from the worker process in the set of
7 distributed worker processes that is assigned the subgraph that contains the logical
8 dependency graph node corresponding to the output rule.

9 102. The medium of embodiment 101, wherein the rule lookup table data structure
10 contains a set of identification tuple to rule data structure mappings.

11 103. The medium of embodiment 102, wherein a rule data structure includes a data
12 field that references an associated concrete atom data structure that implements a
13 forward calculation operation of a generic atom interface to calculate a result.

14 104. The medium of embodiment 103, wherein a rule data structure includes a data
15 field that references a factory function that creates an instance of the associated
16 concrete atom data structure.

17 105. The medium of one of the embodiments 101-104, wherein an identification
18 tuple comprises an ordered list that includes a rule's namespace and rule name.

19 106. The medium of one of the embodiments 101-104, wherein an identification
20 tuple comprises a hash value calculated based on an ordered list that includes a rule's
21 namespace and rule name.

1 107. The medium of one of the embodiments 101-106, wherein a single rule exists
2 in the set of matching rules, and the single rule is selected as the best matching rule.

3 108. The medium of one of the embodiments 101-106, wherein multiple rules exist
4 in the set of matching rules, and a rule with the minimum inheritance distance is
5 selected as the best matching rule.

6 109. The medium of embodiment 108, wherein multiple rules with the minimum
7 inheritance distance exist, and the best matching rule is selected using a tie breaker
8 technique based on one of: position of rule parameters, types of rule parameters.

9 110. The medium of one of the embodiments 101-109, wherein the instructions to
10 (e) generate the logical dependency graph data structure further comprise
11 instructions to:

12 (e1) determine, via the processor, a logical dependency graph node identifier
13 for the best matching rule;

14 (e2) either, upon determining, via the processor, that the logical dependency
15 graph node identifier does not yet exist in the logical dependency graph data
16 structure:

17 add, via the processor, a new logical dependency graph node,
18 corresponding to the best matching rule, to the logical dependency graph data
19 structure, wherein the new logical dependency graph node is assigned the logical
20 dependency graph node identifier;

1 upon determining, via the processor, that the current rule is not the
2 output rule, add a logical dependency graph edge between the new logical dependency
3 graph node and a logical dependency graph node corresponding to the best matching
4 rule's dependent rule;

5 determine, via the processor, a set of precedent rules for the best
6 matching rule, wherein the best matching rule is a dependent rule for precedent rules
7 in the set of precedent rules; and

8 process, via the processor, each respective precedent rule in the set of
9 precedent rules by: setting the current rule to the respective precedent rule, and
10 repeating elements (c)-(e); and

11 return, via the processor, to processing precedent rules of the best
12 matching rule's dependent rule until precedent rules for the output rule are
13 processed;

14 (e3) or, upon determining, via the processor, that the logical dependency graph
15 node identifier already exists in the logical dependency graph data structure:

16 add, via the processor, a logical dependency graph edge between an
17 existing logical dependency graph node, associated with the logical dependency graph
18 node identifier, and a logical dependency graph node corresponding to the best
19 matching rule's dependent rule; and

20 return, via the processor, to processing precedent rules of the best
21 matching rule's dependent rule.

1 111. The medium of one of the embodiments 101-110, wherein a logical dependency
2 graph node is a data structure that includes a data field that that references a
3 corresponding rule's identification tuple.

4 112. The medium of one of the embodiments 101-111, wherein a logical dependency
5 graph edge is a data field in a logical dependency graph node that refers to another
6 logical dependency graph node.

7 113. The medium of one of the embodiments 101-112, wherein the execution
8 complexity gauge value is determined based on at least one of: the number of nodes
9 in the logical dependency graph, the sum of output tensor sizes of logical dependency
10 graph nodes, network data transfer size associated with the logical dependency
11 graph, computation time of the requested calculation.

12 114. The medium of one of the embodiments 101-113, wherein the cardinality of the
13 set of distributed worker processes is determined based on a specified fixed number
14 of available worker processes.

15 115. The medium of one of the embodiments 101-113, wherein the cardinality of the
16 set of distributed worker processes is determined based on the execution complexity
17 gauge value and a scaling parameter value, wherein the scaling parameter value is
18 determined based on at least one of: target processing time, target available memory,
19 target cost, target data transfer size.

1 116. The medium of embodiment 115, wherein the cardinality of the set of
2 distributed worker processes is further determined based on an average worker
3 execution capability.

4 117. The medium of one of the embodiments 101-116, wherein the number of nodes
5 in a subgraph assigned to each respective worker process is determined based on the
6 respective worker's worker execution capability, wherein worker execution capability
7 is determined based on at least one of: processor speed, memory size, network speed.

8 118. The medium of one of the embodiments 101-117, wherein at least some of
9 logical dependency graph nodes in the generated logical dependency graph data
10 structure are duplicated in multiple subgraphs that are assigned to different worker
11 processes.

12 119. The medium of one of the embodiments 101-118, wherein the generated logical
13 dependency graph data structure includes a packaged logical dependency graph node
14 that utilizes an internal logical dependency graph data structure, wherein the
15 internal logical dependency graph data structure is constructed using an associated
16 packaged atom data structure.

17 120. The medium of embodiment 119, wherein the internal logical dependency
18 graph data structure is in the same memory space as the worker process in the set of
19 distributed worker processes that is assigned the subgraph that contains the
20 packaged logical dependency graph node.

1 121. The medium of embodiment 119, wherein the internal logical dependency
2 graph data structure is distributed across multiple worker processes in the set of
3 distributed worker processes.

4 122. The medium of one of the embodiments 101-121, further comprising:

5 the medium stores processor-issuable instructions to:

6 provide a graph representation of the logical dependency graph data
7 structure;

8 obtain a user selection of a target node in the graph representation;

9 determine a target logical dependency graph node in the logical
10 dependency graph data structure corresponding to the user selected target node in
11 the graph representation;

12 obtain a computation result of the target logical dependency graph node
13 from the worker process in the set of distributed worker processes that is assigned
14 the subgraph that contains the target logical dependency graph node; and

15 provide the computation result for display.

16 123. The medium of embodiment 122, wherein the target node in the graph
17 representation includes a visual indicator that specifies the worker process in the set
18 of distributed worker processes that is assigned the subgraph that contains the target
19 logical dependency graph node.

20 124. The medium of embodiment 123, wherein the visual indicator is at least one
21 of: color, color pattern, shape, or border style of the target node.

1 125. The medium of embodiment 122, wherein each node in the graph
2 representation includes a visual indicator that specifies whether the respective node's
3 precedent nodes are displayed.

4 126. The medium of embodiment 125, wherein the visual indicator is at least one
5 of: color, color pattern, shape, or border style of the respective node.

6 127. The medium of one of the embodiments 122-126, further comprising:

7 the medium stores processor-issuable instructions to:

8 obtain a user command to view state replication code, wherein execution
9 of state replication code on a remote client replicates state of an associated logical
10 dependency graph node on the remote client;

11 determine state replication code associated with the target logical
12 dependency graph node; and

13 generate a display containing the determined state replication code.

14 128. The medium of one of the embodiments 122-127, further comprising:

15 the medium stores processor-issuable instructions to:

16 obtain a user command to switch graph display mode;

17 provide a second graph representation of a rule dependency graph data
18 structure corresponding to the logical dependency graph data structure;

19 obtain a user selection of a second target node in the second graph
20 representation; and

1 provide display information associated with the second target node,
2 wherein the display information includes a list of logical dependency graph nodes
3 associated with the second target node and metadata associated with the second
4 target node.

5 129. The medium of one of the embodiments 101-121, further comprising:

6 the medium stores processor-issuable instructions to:

7 provide a graph representation of the logical dependency graph data
8 structure;

9 obtain a user selection of a target node in the graph representation;

10 determine a target logical dependency graph node in the logical
11 dependency graph data structure corresponding to the user selected target node in
12 the graph representation; and

13 provide a second graph representation of the target logical dependency
14 graph node's internal logical dependency graph data structure upon determining that
15 the target logical dependency graph node is a packaged atom.

16 130. The medium of embodiment 129, further comprising:

17 the medium stores processor-issuable instructions to:

18 obtain a user command to switch graph display mode; and

1 provide a third graph representation of a rule dependency graph data
2 structure corresponding to the target logical dependency graph node's internal logical
3 dependency graph data structure.

4 131. The medium of embodiment 130, wherein the third graph representation
5 further includes a graph representation of a rule dependency graph data structure
6 corresponding to the logical dependency graph data structure.

7 132. The medium of one of the embodiments 130-131, wherein nested graph
8 representations of rule dependency graph data structures include navigation links
9 that facilitate navigation among the nested graph representations.

10 133. The medium of one of the embodiments 101-132, wherein at least one of: the
11 best matching rule, a precedent rule of the best matching rule, and a precedent rule
12 of another precedent rule utilizes dynamic precedents.

13 134. The medium of one of the embodiments 101-133, wherein at least one of: the
14 best matching rule, a precedent rule of the best matching rule, and a precedent rule
15 of another precedent rule corresponds to a rule entry in the rule lookup table data
16 structure for a cloned namespace.

17 135. The medium of one of the embodiments 101-134, wherein at least one of: the
18 best matching rule, a precedent rule of the best matching rule, and a precedent rule
19 of another precedent rule is an overriding rule in a cloned namespace determined via
20 a namespace lookup table data structure.

21

1

2 201. A processor-implemented distributed system generating rule compiler method,
3 comprising executing processor-implemented instructions to:

4 (a) obtain, via the processor, a distributed system execution request data
5 structure;

6 (b) determine, via the processor, an output rule and the output rule's set of rule
7 parameters specified via the distributed system execution request data structure for
8 a requested calculation, wherein a current rule is initialized to the output rule;

9 (c) query, via the processor, a rule lookup table data structure to determine a
10 set of matching rules, corresponding to the current rule, based on the current rule's
11 identification tuple;

12 (d) select, via the processor, the best matching rule from the set of matching
13 rules, wherein the best matching rule's set of rule parameters best corresponds to the
14 current rule's set of rule parameters;

15 (e) generate, via the processor, a logical dependency graph data structure by
16 adding logical dependency graph nodes and logical dependency graph edges
17 corresponding to the best matching rule, precedent rules of the best matching rule,
18 and precedent rules of each precedent rule;

19 (f) determine, via the processor, an execution complexity gauge value
20 associated with the generated logical dependency graph data structure;

1 (g) determine, via the processor, a set of distributed worker processes to be
2 utilized to execute the requested calculation;

3 (h) divide, via the processor, the generated logical dependency graph data
4 structure into a set of subgraphs based on the execution complexity gauge value,
5 wherein each worker process in the set of distributed worker processes is assigned a
6 subgraph from the set of subgraphs;

7 (i) initialize, via the processor, each respective worker process in the set of
8 distributed worker processes with the subgraph assigned to the respective worker
9 process;

10 (j) coordinate, via the processor, execution of the requested calculation by
11 worker processes in the set of distributed worker processes, wherein each respective
12 worker process calculates results for logical dependency graph nodes in the subgraph
13 assigned to the respective worker process; and

14 (k) obtain, via the processor, a computation result of the logical dependency
15 graph node corresponding to the output rule, from the worker process in the set of
16 distributed worker processes that is assigned the subgraph that contains the logical
17 dependency graph node corresponding to the output rule.

18 202. The method of embodiment 201, wherein the rule lookup table data structure
19 contains a set of identification tuple to rule data structure mappings.

1 203. The method of embodiment 202, wherein a rule data structure includes a data
2 field that references an associated concrete atom data structure that implements a
3 forward calculation operation of a generic atom interface to calculate a result.

4 204. The method of embodiment 203, wherein a rule data structure includes a data
5 field that references a factory function that creates an instance of the associated
6 concrete atom data structure.

7 205. The method of one of the embodiments 201-204, wherein an identification tuple
8 comprises an ordered list that includes a rule's namespace and rule name.

9 206. The method of one of the embodiments 201-204, wherein an identification tuple
10 comprises a hash value calculated based on an ordered list that includes a rule's
11 namespace and rule name.

12 207. The method of one of the embodiments 201-206, wherein a single rule exists in
13 the set of matching rules, and the single rule is selected as the best matching rule.

14 208. The method of one of the embodiments 201-206, wherein multiple rules exist
15 in the set of matching rules, and a rule with the minimum inheritance distance is
16 selected as the best matching rule.

17 209. The method of embodiment 208, wherein multiple rules with the minimum
18 inheritance distance exist, and the best matching rule is selected using a tie breaker
19 technique based on one of: position of rule parameters, types of rule parameters.

1 210. The method of one of the embodiments 201-209, wherein the instructions to (e)
2 generate the logical dependency graph data structure further comprise instructions
3 to:

4 (e1) determine, via the processor, a logical dependency graph node identifier
5 for the best matching rule;

6 (e2) either, upon determining, via the processor, that the logical dependency
7 graph node identifier does not yet exist in the logical dependency graph data
8 structure:

9 add, via the processor, a new logical dependency graph node,
10 corresponding to the best matching rule, to the logical dependency graph data
11 structure, wherein the new logical dependency graph node is assigned the logical
12 dependency graph node identifier;

13 upon determining, via the processor, that the current rule is not the
14 output rule, add a logical dependency graph edge between the new logical dependency
15 graph node and a logical dependency graph node corresponding to the best matching
16 rule's dependent rule;

17 determine, via the processor, a set of precedent rules for the best
18 matching rule, wherein the best matching rule is a dependent rule for precedent rules
19 in the set of precedent rules; and

1 process, via the processor, each respective precedent rule in the set of
2 precedent rules by: setting the current rule to the respective precedent rule, and
3 repeating elements (c)-(e); and

4 return, via the processor, to processing precedent rules of the best
5 matching rule's dependent rule until precedent rules for the output rule are
6 processed;

7 (e3) or, upon determining, via the processor, that the logical dependency graph
8 node identifier already exists in the logical dependency graph data structure:

9 add, via the processor, a logical dependency graph edge between an
10 existing logical dependency graph node, associated with the logical dependency graph
11 node identifier, and a logical dependency graph node corresponding to the best
12 matching rule's dependent rule; and

13 return, via the processor, to processing precedent rules of the best
14 matching rule's dependent rule.

15 211. The method of one of the embodiments 201-210, wherein a logical dependency
16 graph node is a data structure that includes a data field that that references a
17 corresponding rule's identification tuple.

18 212. The method of one of the embodiments 201-211, wherein a logical dependency
19 graph edge is a data field in a logical dependency graph node that refers to another
20 logical dependency graph node.

1 213. The method of one of the embodiments 201-212, wherein the execution
2 complexity gauge value is determined based on at least one of: the number of nodes
3 in the logical dependency graph, the sum of output tensor sizes of logical dependency
4 graph nodes, network data transfer size associated with the logical dependency
5 graph, computation time of the requested calculation.

6 214. The method of one of the embodiments 201-213, wherein the cardinality of the
7 set of distributed worker processes is determined based on a specified fixed number
8 of available worker processes.

9 215. The method of one of the embodiments 201-213, wherein the cardinality of the
10 set of distributed worker processes is determined based on the execution complexity
11 gauge value and a scaling parameter value, wherein the scaling parameter value is
12 determined based on at least one of: target processing time, target available memory,
13 target cost, target data transfer size.

14 216. The method of embodiment 215, wherein the cardinality of the set of
15 distributed worker processes is further determined based on an average worker
16 execution capability.

17 217. The method of one of the embodiments 201-216, wherein the number of nodes
18 in a subgraph assigned to each respective worker process is determined based on the
19 respective worker's worker execution capability, wherein worker execution capability
20 is determined based on at least one of: processor speed, memory size, network speed.

1 218. The method of one of the embodiments 201-217, wherein at least some of logical
2 dependency graph nodes in the generated logical dependency graph data structure
3 are duplicated in multiple subgraphs that are assigned to different worker processes.

4 219. The method of one of the embodiments 201-218, wherein the generated logical
5 dependency graph data structure includes a packaged logical dependency graph node
6 that utilizes an internal logical dependency graph data structure, wherein the
7 internal logical dependency graph data structure is constructed using an associated
8 packaged atom data structure.

9 220. The method of embodiment 219, wherein the internal logical dependency graph
10 data structure is in the same memory space as the worker process in the set of
11 distributed worker processes that is assigned the subgraph that contains the
12 packaged logical dependency graph node.

13 221. The method of embodiment 219, wherein the internal logical dependency graph
14 data structure is distributed across multiple worker processes in the set of distributed
15 worker processes.

16 222. The method of one of the embodiments 201-221, further comprising:

17 executing processor-implemented instructions to:

18 provide a graph representation of the logical dependency graph data
19 structure;

20 obtain a user selection of a target node in the graph representation;

1 determine a target logical dependency graph node in the logical
2 dependency graph data structure corresponding to the user selected target node in
3 the graph representation;

4 obtain a computation result of the target logical dependency graph node
5 from the worker process in the set of distributed worker processes that is assigned
6 the subgraph that contains the target logical dependency graph node; and

7 provide the computation result for display.

8 223. The method of embodiment 222, wherein the target node in the graph
9 representation includes a visual indicator that specifies the worker process in the set
10 of distributed worker processes that is assigned the subgraph that contains the target
11 logical dependency graph node.

12 224. The method of embodiment 223, wherein the visual indicator is at least one of:
13 color, color pattern, shape, or border style of the target node.

14 225. The method of embodiment 222, wherein each node in the graph
15 representation includes a visual indicator that specifies whether the respective node's
16 precedent nodes are displayed.

17 226. The method of embodiment 225, wherein the visual indicator is at least one of:
18 color, color pattern, shape, or border style of the respective node.

19 227. The method of one of the embodiments 222-226, further comprising:

20 executing processor-implemented instructions to:

1 obtain a user command to view state replication code, wherein execution
2 of state replication code on a remote client replicates state of an associated logical
3 dependency graph node on the remote client;

4 determine state replication code associated with the target logical
5 dependency graph node; and

6 generate a display containing the determined state replication code.

7 228. The method of one of the embodiments 222-227, further comprising:

8 executing processor-implemented instructions to:

9 obtain a user command to switch graph display mode;

10 provide a second graph representation of a rule dependency graph data
11 structure corresponding to the logical dependency graph data structure;

12 obtain a user selection of a second target node in the second graph
13 representation; and

14 provide display information associated with the second target node,
15 wherein the display information includes a list of logical dependency graph nodes
16 associated with the second target node and metadata associated with the second
17 target node.

18 229. The method of one of the embodiments 201-221, further comprising:

19 executing processor-implemented instructions to:

1 provide a graph representation of the logical dependency graph data
2 structure;

3 obtain a user selection of a target node in the graph representation;

4 determine a target logical dependency graph node in the logical
5 dependency graph data structure corresponding to the user selected target node in
6 the graph representation; and

7 provide a second graph representation of the target logical dependency
8 graph node's internal logical dependency graph data structure upon determining that
9 the target logical dependency graph node is a packaged atom.

10 230. The method of embodiment 229, further comprising:

11 executing processor-implemented instructions to:

12 obtain a user command to switch graph display mode; and

13 provide a third graph representation of a rule dependency graph data
14 structure corresponding to the target logical dependency graph node's internal logical
15 dependency graph data structure.

16 231. The method of embodiment 230, wherein the third graph representation
17 further includes a graph representation of a rule dependency graph data structure
18 corresponding to the logical dependency graph data structure.

19 232. The method of one of the embodiments 230-231, wherein nested graph
20 representations of rule dependency graph data structures include navigation links
21 that facilitate navigation among the nested graph representations.

1 233. The method of one of the embodiments 201-232, wherein at least one of: the
2 best matching rule, a precedent rule of the best matching rule, and a precedent rule
3 of another precedent rule utilizes dynamic precedents.

4 234. The method of one of the embodiments 201-233, wherein at least one of: the
5 best matching rule, a precedent rule of the best matching rule, and a precedent rule
6 of another precedent rule corresponds to a rule entry in the rule lookup table data
7 structure for a cloned namespace.

8 235. The method of one of the embodiments 201-234, wherein at least one of: the
9 best matching rule, a precedent rule of the best matching rule, and a precedent rule
10 of another precedent rule is an overriding rule in a cloned namespace determined via
11 a namespace lookup table data structure.

12 THE EMBODIMENTS OF THE DSGRCE

13 **[00189]** The entirety of this disclosure (including the written description,
14 figures, claims, abstract, appendices, and/or the like) for DISTRIBUTED SYSTEM
15 GENERATING RULE COMPILER ENGINE APPARATUSES, METHODS,
16 SYSTEMS AND MEDIA shows various embodiments via which the claimed
17 innovations may be practiced. It is to be understood that these embodiments and
18 the features they describe are a representative sample presented to assist in
19 understanding the claimed innovations, and are not exhaustive and/or exclusive.
20 As such, the various embodiments, implementations, examples, and/or the like
21 are deemed non-limiting throughout this disclosure. Furthermore, alternate

1 und-described embodiments may be available (e.g., equivalent embodiments). Such
2 alternate embodiments have not been discussed in detail to preserve space and/or
3 reduce repetition. That alternate embodiments have not been discussed in detail
4 is not to be considered a disclaimer of such alternate und-described embodiments,
5 and no inference should be drawn regarding such alternate und-described
6 embodiments relative to those discussed in detail in this disclosure. It is to be
7 understood that such alternate und-described embodiments may be utilized without
8 departing from the spirit and/or scope of the disclosure. For example, the
9 organizational, logical, physical, functional, topological, and/or the like structures
10 of various embodiments may differ. In another example, the organizational,
11 logical, physical, functional, topological, and/or the like structures of the DSGRCE
12 coordinator, DSGRCE coordinator elements, DSGRCE data stores, DSGRCE
13 components and their subcomponents, capabilities, applications, and/or the like
14 described in various embodiments throughout this disclosure are not limited to a
15 fixed operating order and/or arrangement, instead, all equivalent operating orders
16 and/or arrangements are contemplated by this disclosure. In yet another example,
17 the DSGRCE coordinator, DSGRCE coordinator elements, DSGRCE data stores,
18 DSGRCE components and their subcomponents, capabilities, applications, and/or
19 the like described in various embodiments throughout this disclosure are not
20 limited to serial execution, instead, any number and/or configuration of threads,
21 processes, instances, services, servers, clients, nodes, and/or the like that execute
22 in parallel, concurrently, simultaneously, synchronously, asynchronously, and/or

1 the like is contemplated by this disclosure. Furthermore, it is to be understood
2 that some of the features described in this disclosure may be mutually
3 contradictory, incompatible, inapplicable, and/or the like, and are not present
4 simultaneously in the same embodiment. Accordingly, the various embodiments,
5 implementations, examples, and/or the like are not to be considered limitations on
6 the disclosure as defined by the claims or limitations on equivalents to the claims.

7 **[00190]** This disclosure includes innovations not currently claimed. Applicant
8 reserves all rights in such currently unclaimed innovations including the rights to
9 claim such innovations and to file additional provisional applications,
10 nonprovisional applications, continuation applications, continuation-in-part
11 applications, divisional applications, and/or the like. Further, this disclosure is
12 not limited to the combination of features specified in the claims and includes
13 combinations of features other than those explicitly recited in the claims. It is to
14 be understood that while some embodiments of the DSGRCE discussed in this
15 disclosure have been directed to cloud-based distributed systems, the innovations
16 described in this disclosure may be readily applied to a wide variety of other fields
17 and/or applications.

18

CLAIMS

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21

The following is claimed:

1. A distributed system generating rule compiler apparatus, comprising:
 - a memory;
 - a processor in communication with the memory, and configured to issue a plurality of processing instructions stored in the memory, wherein the processor issues instructions to:
 - (a) obtain, via the processor, a distributed system execution request data structure;
 - (b) determine, via the processor, an output rule and the output rule's set of rule parameters specified via the distributed system execution request data structure for a requested calculation, wherein a current rule is initialized to the output rule;
 - (c) query, via the processor, a rule lookup table data structure to determine a set of matching rules, corresponding to the current rule, based on the current rule's identification tuple;
 - (d) select, via the processor, the best matching rule from the set of matching rules, wherein the best matching rule's set of rule parameters best corresponds to the current rule's set of rule parameters;
 - (e) generate, via the processor, a logical dependency graph data structure by adding logical dependency graph nodes and logical dependency graph

1 edges corresponding to the best matching rule, precedent rules of the best matching
2 rule, and precedent rules of each precedent rule;

3 (f) determine, via the processor, an execution complexity gauge value
4 associated with the generated logical dependency graph data structure;

5 (g) determine, via the processor, a set of distributed worker processes to
6 be utilized to execute the requested calculation;

7 (h) divide, via the processor, the generated logical dependency graph
8 data structure into a set of subgraphs based on the execution complexity gauge value,
9 wherein each worker process in the set of distributed worker processes is assigned a
10 subgraph from the set of subgraphs;

11 (i) initialize, via the processor, each respective worker process in the set
12 of distributed worker processes with the subgraph assigned to the respective worker
13 process;

14 (j) coordinate, via the processor, execution of the requested calculation
15 by worker processes in the set of distributed worker processes, wherein each
16 respective worker process calculates results for logical dependency graph nodes in the
17 subgraph assigned to the respective worker process; and

18 (k) obtain, via the processor, a computation result of the logical
19 dependency graph node corresponding to the output rule, from the worker process in
20 the set of distributed worker processes that is assigned the subgraph that contains
21 the logical dependency graph node corresponding to the output rule.

1 2. The apparatus of claim 1, wherein the rule lookup table data structure contains
2 a set of identification tuple to rule data structure mappings.

3 3. The apparatus of claim 2, wherein a rule data structure includes a data field
4 that references an associated concrete atom data structure that implements a
5 forward calculation operation of a generic atom interface to calculate a result.

6 4. The apparatus of claim 3, wherein a rule data structure includes a data field
7 that references a factory function that creates an instance of the associated concrete
8 atom data structure.

9 5. The apparatus of claim 1, wherein an identification tuple comprises an ordered
10 list that includes a rule's namespace and rule name.

11 6. The apparatus of claim 1, wherein an identification tuple comprises a hash
12 value calculated based on an ordered list that includes a rule's namespace and rule
13 name.

14 7. The apparatus of claim 1, wherein a single rule exists in the set of matching
15 rules, and the single rule is selected as the best matching rule.

16 8. The apparatus of claim 1, wherein multiple rules exist in the set of matching
17 rules, and a rule with the minimum inheritance distance is selected as the best
18 matching rule.

19 9. The apparatus of claim 8, wherein multiple rules with the minimum
20 inheritance distance exist, and the best matching rule is selected using a tie breaker
21 technique based on one of: position of rule parameters, types of rule parameters.

1 10. The apparatus of claim 1, wherein the instructions to (e) generate the logical
2 dependency graph data structure further comprise instructions to:

3 (e1) determine, via the processor, a logical dependency graph node identifier
4 for the best matching rule;

5 (e2) either, upon determining, via the processor, that the logical dependency
6 graph node identifier does not yet exist in the logical dependency graph data
7 structure:

8 add, via the processor, a new logical dependency graph node,
9 corresponding to the best matching rule, to the logical dependency graph data
10 structure, wherein the new logical dependency graph node is assigned the logical
11 dependency graph node identifier;

12 upon determining, via the processor, that the current rule is not the
13 output rule, add a logical dependency graph edge between the new logical dependency
14 graph node and a logical dependency graph node corresponding to the best matching
15 rule's dependent rule;

16 determine, via the processor, a set of precedent rules for the best
17 matching rule, wherein the best matching rule is a dependent rule for precedent rules
18 in the set of precedent rules; and

19 process, via the processor, each respective precedent rule in the set of
20 precedent rules by: setting the current rule to the respective precedent rule, and
21 repeating elements (c)-(e); and

1 return, via the processor, to processing precedent rules of the best
2 matching rule's dependent rule until precedent rules for the output rule are
3 processed;

4 (e3) or, upon determining, via the processor, that the logical dependency graph
5 node identifier already exists in the logical dependency graph data structure:

6 add, via the processor, a logical dependency graph edge between an
7 existing logical dependency graph node, associated with the logical dependency graph
8 node identifier, and a logical dependency graph node corresponding to the best
9 matching rule's dependent rule; and

10 return, via the processor, to processing precedent rules of the best
11 matching rule's dependent rule.

12 11. The apparatus of claim 1, wherein a logical dependency graph node is a data
13 structure that includes a data field that that references a corresponding rule's
14 identification tuple.

15 12. The apparatus of claim 1, wherein a logical dependency graph edge is a data
16 field in a logical dependency graph node that refers to another logical dependency
17 graph node.

18 13. The apparatus of claim 1, wherein the execution complexity gauge value is
19 determined based on at least one of: the number of nodes in the logical dependency
20 graph, the sum of output tensor sizes of logical dependency graph nodes, network

1 data transfer size associated with the logical dependency graph, computation time of
2 the requested calculation.

3 14. The apparatus of claim 1, wherein the cardinality of the set of distributed
4 worker processes is determined based on a specified fixed number of available worker
5 processes.

6 15. The apparatus of claim 1, wherein the cardinality of the set of distributed
7 worker processes is determined based on the execution complexity gauge value and a
8 scaling parameter value, wherein the scaling parameter value is determined based
9 on at least one of: target processing time, target available memory, target cost, target
10 data transfer size.

11 16. The apparatus of claim 15, wherein the cardinality of the set of distributed
12 worker processes is further determined based on an average worker execution
13 capability.

14 17. The apparatus of claim 1, wherein the number of nodes in a subgraph assigned
15 to each respective worker process is determined based on the respective worker's
16 worker execution capability, wherein worker execution capability is determined
17 based on at least one of: processor speed, memory size, network speed.

18 18. The apparatus of claim 1, wherein at least some of logical dependency graph
19 nodes in the generated logical dependency graph data structure are duplicated in
20 multiple subgraphs that are assigned to different worker processes.

1 19. The apparatus of claim 1, wherein the generated logical dependency graph
2 data structure includes a packaged logical dependency graph node that utilizes an
3 internal logical dependency graph data structure, wherein the internal logical
4 dependency graph data structure is constructed using an associated packaged atom
5 data structure.

6 20. The apparatus of claim 19, wherein the internal logical dependency graph data
7 structure is in the same memory space as the worker process in the set of distributed
8 worker processes that is assigned the subgraph that contains the packaged logical
9 dependency graph node.

10 21. The apparatus of claim 19, wherein the internal logical dependency graph data
11 structure is distributed across multiple worker processes in the set of distributed
12 worker processes.

13 22. The apparatus of claim 1, further comprising:

14 the processor issues instructions to:

15 provide a graph representation of the logical dependency graph data
16 structure;

17 obtain a user selection of a target node in the graph representation;

18 determine a target logical dependency graph node in the logical
19 dependency graph data structure corresponding to the user selected target node in
20 the graph representation;

1 obtain a computation result of the target logical dependency graph node
2 from the worker process in the set of distributed worker processes that is assigned
3 the subgraph that contains the target logical dependency graph node; and
4 provide the computation result for display.

5 23. The apparatus of claim 22, wherein the target node in the graph representation
6 includes a visual indicator that specifies the worker process in the set of distributed
7 worker processes that is assigned the subgraph that contains the target logical
8 dependency graph node.

9 24. The apparatus of claim 23, wherein the visual indicator is at least one of: color,
10 color pattern, shape, or border style of the target node.

11 25. The apparatus of claim 22, wherein each node in the graph representation
12 includes a visual indicator that specifies whether the respective node's precedent
13 nodes are displayed.

14 26. The apparatus of claim 25, wherein the visual indicator is at least one of: color,
15 color pattern, shape, or border style of the respective node.

16 27. The apparatus of claim 22, further comprising:

17 the processor issues instructions to:

18 obtain a user command to view state replication code, wherein execution
19 of state replication code on a remote client replicates state of an associated logical
20 dependency graph node on the remote client;

1 determine state replication code associated with the target logical
2 dependency graph node; and

3 generate a display containing the determined state replication code.

4 28. The apparatus of claim 22, further comprising:

5 the processor issues instructions to:

6 obtain a user command to switch graph display mode;

7 provide a second graph representation of a rule dependency graph data
8 structure corresponding to the logical dependency graph data structure;

9 obtain a user selection of a second target node in the second graph
10 representation; and

11 provide display information associated with the second target node,
12 wherein the display information includes a list of logical dependency graph nodes
13 associated with the second target node and metadata associated with the second
14 target node.

15 29. The apparatus of claim 1, further comprising:

16 the processor issues instructions to:

17 provide a graph representation of the logical dependency graph data
18 structure;

19 obtain a user selection of a target node in the graph representation;

1 determine a target logical dependency graph node in the logical
2 dependency graph data structure corresponding to the user selected target node in
3 the graph representation; and

4 provide a second graph representation of the target logical dependency
5 graph node's internal logical dependency graph data structure upon determining that
6 the target logical dependency graph node is a packaged atom.

7 30. The apparatus of claim 29, further comprising:

8 the processor issues instructions to:

9 obtain a user command to switch graph display mode; and

10 provide a third graph representation of a rule dependency graph data
11 structure corresponding to the target logical dependency graph node's internal logical
12 dependency graph data structure.

13 31. The apparatus of claim 30, wherein the third graph representation further
14 includes a graph representation of a rule dependency graph data structure
15 corresponding to the logical dependency graph data structure.

16 32. The apparatus of claim 30, wherein nested graph representations of rule
17 dependency graph data structures include navigation links that facilitate navigation
18 among the nested graph representations.

19 33. The apparatus of claim 1, wherein at least one of: the best matching rule, a
20 precedent rule of the best matching rule, and a precedent rule of another precedent
21 rule utilizes dynamic precedents.

1 34. The apparatus of claim 1, wherein at least one of: the best matching rule, a
2 precedent rule of the best matching rule, and a precedent rule of another precedent
3 rule corresponds to a rule entry in the rule lookup table data structure for a cloned
4 namespace.

5 35. The apparatus of claim 1, wherein at least one of: the best matching rule, a
6 precedent rule of the best matching rule, and a precedent rule of another precedent
7 rule is an overriding rule in a cloned namespace determined via a namespace lookup
8 table data structure.

9 36. A distributed system generating rule compiler processor-readable non-
10 transitory physical medium storing processor-issuable instructions to:

11 (a) obtain, via the processor, a distributed system execution request data
12 structure;

13 (b) determine, via the processor, an output rule and the output rule's set of rule
14 parameters specified via the distributed system execution request data structure for
15 a requested calculation, wherein a current rule is initialized to the output rule;

16 (c) query, via the processor, a rule lookup table data structure to determine a
17 set of matching rules, corresponding to the current rule, based on the current rule's
18 identification tuple;

19 (d) select, via the processor, the best matching rule from the set of matching
20 rules, wherein the best matching rule's set of rule parameters best corresponds to the
21 current rule's set of rule parameters;

1 (e) generate, via the processor, a logical dependency graph data structure by
2 adding logical dependency graph nodes and logical dependency graph edges
3 corresponding to the best matching rule, precedent rules of the best matching rule,
4 and precedent rules of each precedent rule;

5 (f) determine, via the processor, an execution complexity gauge value
6 associated with the generated logical dependency graph data structure;

7 (g) determine, via the processor, a set of distributed worker processes to be
8 utilized to execute the requested calculation;

9 (h) divide, via the processor, the generated logical dependency graph data
10 structure into a set of subgraphs based on the execution complexity gauge value,
11 wherein each worker process in the set of distributed worker processes is assigned a
12 subgraph from the set of subgraphs;

13 (i) initialize, via the processor, each respective worker process in the set of
14 distributed worker processes with the subgraph assigned to the respective worker
15 process;

16 (j) coordinate, via the processor, execution of the requested calculation by
17 worker processes in the set of distributed worker processes, wherein each respective
18 worker process calculates results for logical dependency graph nodes in the subgraph
19 assigned to the respective worker process; and

20 (k) obtain, via the processor, a computation result of the logical dependency
21 graph node corresponding to the output rule, from the worker process in the set of

1 distributed worker processes that is assigned the subgraph that contains the logical
2 dependency graph node corresponding to the output rule.

3 37. A processor-implemented distributed system generating rule compiler method,
4 comprising executing processor-implemented instructions to:

5 (a) obtain, via the processor, a distributed system execution request data
6 structure;

7 (b) determine, via the processor, an output rule and the output rule's set of rule
8 parameters specified via the distributed system execution request data structure for
9 a requested calculation, wherein a current rule is initialized to the output rule;

10 (c) query, via the processor, a rule lookup table data structure to determine a
11 set of matching rules, corresponding to the current rule, based on the current rule's
12 identification tuple;

13 (d) select, via the processor, the best matching rule from the set of matching
14 rules, wherein the best matching rule's set of rule parameters best corresponds to the
15 current rule's set of rule parameters;

16 (e) generate, via the processor, a logical dependency graph data structure by
17 adding logical dependency graph nodes and logical dependency graph edges
18 corresponding to the best matching rule, precedent rules of the best matching rule,
19 and precedent rules of each precedent rule;

20 (f) determine, via the processor, an execution complexity gauge value
21 associated with the generated logical dependency graph data structure;

1 (g) determine, via the processor, a set of distributed worker processes to be
2 utilized to execute the requested calculation;

3 (h) divide, via the processor, the generated logical dependency graph data
4 structure into a set of subgraphs based on the execution complexity gauge value,
5 wherein each worker process in the set of distributed worker processes is assigned a
6 subgraph from the set of subgraphs;

7 (i) initialize, via the processor, each respective worker process in the set of
8 distributed worker processes with the subgraph assigned to the respective worker
9 process;

10 (j) coordinate, via the processor, execution of the requested calculation by
11 worker processes in the set of distributed worker processes, wherein each respective
12 worker process calculates results for logical dependency graph nodes in the subgraph
13 assigned to the respective worker process; and

14 (k) obtain, via the processor, a computation result of the logical dependency
15 graph node corresponding to the output rule, from the worker process in the set of
16 distributed worker processes that is assigned the subgraph that contains the logical
17 dependency graph node corresponding to the output rule.