

APPLICATIONS FOR LOW CODE, INTERNET OF THINGS, AND HIGHLY DISTRIBUTED ENVIRONMENTS

CLAIM OF PRIORITY

[0001] This application claims priority under 35 U.S.C. §119(e) to provisional U.S. Patent Application Serial No. 63/437,300, filed on January 05, 2023, and 63/528,824, filed on July 25, 2023, the entire contents of each of which are hereby incorporated by reference.

TECHNICAL FIELD

[0002] This disclosure generally relates to data processing systems.

BACKGROUND

[0003] Internet of Things (IoT) systems are hard to design, implement, deploy and maintain. These IoT problems are the source of a reported 30% failure rate of IoT projects in the trial phase. Simultaneously, the construction of very large-scale distributed systems for applications has lead to the recognition of several similar core problems with these systems including: dynamic parameter adjustment for improved performance, dynamically repair problems to increase availability, scale up of systems by two orders of magnitude that create scheduling problems, cloud deployments that create complex configuration problems, heterogeneous hardware, and new data provenance requirements. Finally, "no-code" or "low-code" platforms have become a popular way to deliver prepackaged patterns for common business operations (customer relationship management, websites for retail distributors, robotic process automation, etc.).

SUMMARY

[0004] This disclosure describes systems and processes for generating applications for different platform types including low-code, internet of things, and highly distributed environments. An application is defined that includes projections of an application along different dimensions. Once each projection is defined, the application is created by combining the projections in a predetermined way. The results show that applications constructed in this way are easier to design, construct, deploy, and maintain. Much of the operational capabilities of the system use the theory itself to add new functionality.

[0005] Modern computing applications are a complex combination of user interfaces, application code, and infrastructure components. This complex combination makes

applications hard to design, construct, deploy and maintain. To overcome these issues, the systems and processes described apply a methodology of separation of concerns of what an application computes from how an application computes. For three example classes of applications: Internet of Things, highly distributed systems, and low-code business applications, this disclosure describes describe a fundamental, formal theory of the application that constitutes the basis for a solution for all three classes.

[0006] A theory of the application is a normal form of an application that describes the constituent parts of the application - the entities, data, operations, communication, etc. - that form the core specification of an application. This normal form is the basis on which all the other required application development capabilities (e.g. testing, optimization, parallelization and distributed execution, deployment, security management) are built.

[0007] The role of the normal form of an application is similar in nature to that of the schema of a database. The databases schema is a formal representation of structure of information in the database, and so is our normal form. The database schema is the shared representation of the application between domain specialists and engineers. The schema is the middle ground they both understand and use to communicate. Functionality is defined in terms of the schema (e.g. queries, indexes, views, triggers, access patterns, parallelization) so the schema is the fundamental core of the database. The normal form serves the same purpose as a database schema for the case of applications, for the same two purposes: as a common language, and as a core for which software engineering tasks are defined.

[0008] An application specification is normalized, or decomposed into essential pieces, to guarantee a set of goals including: the elimination of redundant information in design of an application; the minimization of the number of architecture decisions taken by the application designer; a definition of a complete separation of concerns between various aspects of the application; and opportunity for reuse.

[0009] Traditional applications inseparably mix the specification and the code. For example, in a simple IoT system, there are sensors (cameras), actuators (alerting security personnel) and an application that mixes specification "if an unrecognized person enters a secure zone, notify security personnel" with code "camera @128.256.0.0 contacts the messaging service @98.128.1.1 ... etc." (see system 100 of Figure 1). The normal form separate separates the "what" application specification from the "how" code. The specification provides information to a runtime system that orchestrates the interaction between the sensors, "how" code, and actuators (see system 200 of Figure 2).

[0010] The first essential step in normalization separates between the specification of an application (the what), from the execution of an application (the how). For example, "upon entrance in a hospital a picture of the entering person is taken, and image recognition is attempted on that photo, then alerts are raised in certain cases" describes what the application intends to do. "Cameras are controlled by RaspberryPis and image recognition is executed in the cloud" describes the how the application will be actually deployed.

[0011] Second, the logical structure of the application is separated from code. The flow of information between various processing points is expressed separately, in a formal and code-free form. The flow of the information (e.g. "picture is taken by camera, then sent to image recognition, then the enriched image is passed through a set of rules to detect possible alarm causes") is described explicitly as data, and is independent from how each processing point is implemented. The clean separation between the data/formal part of the application describing the structure of the flow of information, from the actual code that implements each processing point is the crucial point in our design. This separation allows the clean injection of various functionality into the flow of information.

[0012] Finally, the code is diced into stand-alone and reusable components with the simplest (but not simpler than necessary) interface. Examples are image recognition, automatas, data enrichment, generic ML pattern matching, etc. Each components can be designed, implemented, optimized and tested separately. Such code components are heavily parameterized for flexibility, then standardized and reused as much as possible as building blocks for creating applications.

[0013] In this framework, designing an application results in a simple methodology: (a) decide a particular flow of information (i.e. the dataflow) and (b) assemble such (most likely pre-existing) code components to support the dataflow, and (c) add any domain specific code. Given such a high level declaration of an application, the supporting (optimized) runtime can be automatically generated. Automatic generation avoids the many problems of synchronizing the application with the infrastructure in a distributed system. This methodology provides a viable strategy to build high quality software application in an rapid "assembly line" way.

[0014] Low-code and No-code

[0015] In the lifetime of the application there are many stakeholders, each category with their own skill set, their knowledge base and their own particular goals that need to be reconciled. The normalized application specification constitutes the single shared representation of the application, and it is shared with most stake holders: the domain specialists refer to it, APIs and real code can be generated from it for application developers

and hardware engineers, financial costs can be estimated based on it, etc. Tools are built around the normal form, emphasizing for each stakeholder the aspect of the application that is most important to them.

[0016] From a pure software development point of view, the normalized application specification is the basis for tools that simplify multiple costly engineering tasks: code generation, automatic testing, automatic optimizations, automatic parallelization and distribution, automatic updates, failure analysis, security risks checks, functionality injection, automatic adjustment for testing, performance analysis via benchmarks or simulation, etc. More broadly, standardization of components is of uttermost importance if we are pursuing transforming application development from a manual craft into a streamlined industry. The data processing systems described herein enable a clear shape for such components, that can be combined and re-combined to assemble various applications.

[0017] **Internet of Things**

[0018] IoT applications, where sensors are constantly emitting data that triggers various computations, are the poster child for the type of applications for which we are interested: event-based dataflow architectures. In this architecture, computation is always triggered by an event. The source of the event can be by human input, by sensor data gathered from the real world, or by simply raising a software signal. Once triggered, the computation results in another series of events that are further processed by other software components. The application specification models precisely this type of dataflow applications: a standard form in IoT but used recently for most modern (even non-IoT) applications. Finally, IoT highlights common pattern in all modern applications: the execution on a highly distributed and highly heterogeneous computing environments.

[0019] **Highly Distributed and Heterogeneous Environments**

[0020] A single architecture generally does not meet all application needs. Most applications mix and match centralized (in the “cloud”, private or public) and highly distributed processing environments that range from local servers, to mobile devices, to embedded controllers. Such computing points can be linked through various network protocols, making compromises between various dimensions: cost, bandwidth, power, security.

[0021] The normal form of an application that is described in this work provides the framework on which the parallelization of computation on such a distributed platform can be designed (ideally in an automated way) and performed. Having a declarative specification of processing points and an abstract specification of the available computational resources lays the foundation of a general “distributed execution plan”.

[0022] The systems and processes herein can include one or more of the implementations or embodiments described in the examples section below.

[0023] The details of one or more embodiments of these systems and methods are set forth in the accompanying drawings and the description to be presented. Other features, objects, and advantages of these systems and methods are apparent from the description and drawings, and from the claims.

BRIEF DESCRIPTION OF THE DRAWINGS

[0024] Fig. 1 shows an example application for a data processing system.

[0025] Fig. 2 shows an example system in which an application specification orchestrates data processing.

[0026] Fig. 3 shows an example graph of entities linked through the "connected" relationship.

[0027] Fig. 4 shows an example graph of entities linked through the "has_a" relationship.

[0028] Fig. 5 shows an example dataflow graph of a running example.

[0029] Fig. 6 shows the instance graph of the running example with only the has_a relationship.

[0030] Fig. 7 shows an example graph including relationships between the operators, ontology, dataflow, instances and the context, specification and runtime system.

[0031] Fig. 8 shows an example runtime graph of the running example.

[0032] Fig. 9 shows an example tool chain from application specification to execution.

[0033] Fig. 10 shows an example partial view of the entities file for the running example and is missing the declaration of the "connected" relationship.

[0034] Fig. 11 shows an example view of specification and implementation details of the CameraSensor operator.

[0035] Fig. 12 shows an example combined description of the dataflow graph and the anchored dataflow graph.

[0036] Fig. 13 shows example instances file that includes a description of the instances graph on which the application operates.

[0037] Fig. 14 shows an example class structure.

[0038] Fig. 15 shows an example process for processing a send and a process.

[0039] Fig. 16 shows an example example operator.

[0040] Fig. 17 shows an example autonoma graph.

- [0041] Fig. 18 shows an example process of calls that implement call/compute functionality via send/process functionality.
- [0042] Fig. 19 shows an example of a Caller superclass.
- [0043] Fig. 20 shows an example sketch of the Callee class.
- [0044] Fig. 21 shows an example task that declares the provenance property.
- [0045] Fig. 22 shows an image of a domain expert modeling the specification using a design tool.
- [0046] Fig. 23 shows an image that represents the domain expert modeling the specification using a design tool.
- [0047] Fig. 24 shows an example validate process.
- [0048] Fig. 25 shows an example injection process.
- [0049] Figure 26 shows an example parameterization process.
- [0050] Fig. 27 shows an example syntax for partitioning the set of tasks in the dataflow into a set of task clusters.
- [0051] Fig. 28 shows an example cluster organization of tasks in the running example
- [0052] Fig. 29 shows an example CPU specification graph including of the active entity graph and the connected relation in the running example.
- [0053] Fig. 30 shows an example network communication connectivity in the running example.
- [0054] Fig. 31 shows an example cluster placement decision for the running example.
- [0055] Fig. 32 shows an example placement of clusters in the CPU graph for the running example.
- [0056] Fig. 33 shows an example syntax to describe a cluster communication specification.
- [0057] Fig. 34 shows an example broker communication between placed clusters in the running example.
- [0058] Fig. 35 shows an example broker communication.
- [0059] Fig. 36 shows an example distributed execution plan of the runtime example.
- [0060] Fig. 37 shows an example conceptualization of the instantiated distributed execution plan of the runtime example.
- [0061] Fig. 38 shows an example distributed execution plan of the modified runtime example.
- [0062] Fig. 39 shows an example conceptualization of the modified instantiated distributed execution plan of the runtime example of Fig. 38.

- [0063] Fig. 40 shows an example distributed execution plan of the modified runtime example.
- [0064] Fig. 41 shows an example small portion of the centralized dataflow graph.
- [0065] Fig. 42 shows an example result of editing the runtime graph of Figure 41.
- [0066] Fig. 43 shows an example specification and implementation details of the CameraSensor operator.
- [0067] Fig. 44 shows an example parameter set for an automata that uses the AutomataTemplate.
- [0068] Fig. 45 shows example Python classify and state methods of the example automata that uses the AutomataTemplate.
- [0069] Fig. 46 shows an example sketch of the transition method of AutomataTemplate in the Python library.
- [0070] Fig. 47 shows an example sketch of the Database operator implementation.
- [0071] Fig. 48 shows an example sketch of the Data Enrichment operator implementation.
- [0072] Fig. 49 shows an example included file generator__distributions.yaml for parameters.
- [0073] Fig. 50 shows example parameters for the DataGenerator operator.
- [0074] Fig. 51. shows an example sketch of the main processing of the EventConditionAction template.
- [0075] Fig. 52 shows an example sketch of the query method of user code that inherits from StreamingTemplate.
- [0076] Fig. 53 shows an example sketch of the CameraSensorPi operator.
- [0077] Fig. 54 shows an example sketch of the user logging declaration.
- [0078] Fig. 55 shows an example of the dashboard declaration.
- [0079] Fig. 56 shows example yaml describing merging the cluster one and cluster two.
- [0080] Fig. 57 shows example yaml describing splitting a cluster.
- [0081] Fig. 58 shows an example placement update.
- [0082] Fig. 59 shows an example a communication update.
- [0083] Fig. 60 shows example yaml describing adding a new task connection.
- [0084] Fig. 61 shows example yaml describing removing a task connection.
- [0085] Fig. 62 shows example yaml describing adding a new task.
- [0086] Fig. 63 shows example yaml describing removing a task.
- [0087] Fig. 64 shows example yaml describing changing an operator parameter value.
- [0088] Fig. 65 shows example yaml describing changing a task parameter value.

- [0089] Fig. 66 shows example yaml describing changing a task system parameter.
- [0090] Fig. 67 shows an example of adding a new entity update.
- [0091] Fig. 68 shows an example of removing an entity.
- [0092] Fig. 69. shows example yaml describing changing an operator implementation.
- [0093] Fig. 70 shows an example of adding a new port to an operator.
- [0094] Fig. 71 shows an example of removing a port from an operator.
- [0095] Fig. 72 shows example yaml describing adding a new subtree.
- [0096] Fig. 73 shows example yaml describing deleting a subtree.
- [0097] Fig. 74 shows example yaml describing changing a task-instance parameter value.
- [0098] Fig. 75 shows an example architecture of a simulation optimization.
- [0099] Fig. 76 shows an example of targeting tasks for zoom debugging.
- [0100] Fig. 77 shows an example of logging input and output for zoom debugging.
- [0101] Fig. 78 shows an example of replaying targeted tasks from logged input and output.
- [0102] Figs. 79-83 show example processes.
- [0103] Fig. 84 is a block diagram of an example computer system used to provide computational functionalities associated with described algorithms, methods, functions, processes, flows, and procedures described in the present disclosure.

DETAILED DESCRIPTION

[0104] Normal Form Of Applications

[0105] This section describes the normal form for applications. This process is driven by the following goals: eliminate the redundancy of information in designing and implementing an application, minimize of the number of decisions taken by the application designer, achieve a complete separation of concerns, and maximize opportunity for reuse.

[0106] The normal form is the result of unraveling and decomposing an application into its core individual components. Unlike traditional software engineering method of splitting a large software application into software modules, a component in our definition is something completely different. A component describes the projection of the application on one of the five dimension on which the application is being built: the ontology of the real world objects on which the application is applied, the data being the set of data structures that the application is handling, the flow of processing and information/data passing, the code of each processing point, and the instance of the ontology on which the application is applied. Together, the projections create the application as a integration of those projections. The integration is accomplished by combining the various projections in a strict mathematical formulation. Designing in isolation each projection is infinitely simpler than designing and coding in a non-declarative programming language that mingles the various aspects of an application together.

[0107] Running Example

[0108] To illustrate the architecture, a running example is used throughout the disclosure. The example focuses on an application to detect potential COVID situations with the following requirements as defined by a hypothetical domain expert: If any of the following conditions occur: More people in a room than its capacity, Some unknown person is in the room (based on a preexisting list of known people), then perform the actions: Notify an security personnel that a problem may exist.

[0109] Application Context

[0110] Each application is developed in a certain context that is characteristic to its particular vertical domain. The context of an application is defined by three of the projections mentioned above: the entity ontology specifies the real world entities on which the application operates, and the relationships among them, the data types library that includes all data types manipulated by the application, the operators algebra specifies the set of available code components of the application. Among the three components, the data type library can be thought of as a database schema for application domain information.

[0111] Real World Entities Ontology

[0112] An application performs operations pertaining to some existing entities, whether physical (e.g. rooms, doors) or virtual (e.g. payment). The application designer defines the ontology of the real world entities the application operates on, and the relationship between them, as follows. The real world entities ontology (the entities ontology for short) is a graph composed of: A set of entities E as nodes; A set of relationship identifiers R ; and a set of directed edges $(e1, e2)$ between pairs of entities, where $e1 \in E$ and $e2 \in E$, and each edge is labeled with a relationship identifier $r \in R$.

[0113] In the running example, the entity ontology E is the graph 400 (Figure 4) where: the nodes are the entities $E = \{Building, Room, Door\}$, and the relationship labels are $R = \{has_a, connected\}$, and the edges are (Building, Room, "has_a"), (Room, Door, "has_a"). In addition, the relationship "connected" is calculated as the transitive closure of the "has_a" relationship, unioned with the inverse of the transitive closure, unioned with the identity relationship, as shown in Table 1, Figure 3, and Figure 4. Fig. 3 shows a graph 300 of entities linked through the "connected" relationship. Fig. 4 shows a graph of 400 entities linked through the "has_a" relationship.

[0114] Table 1. The edges in the entity graph in the running example.

From	To	Relationship
Building	Room	has_a
Room	Door	has_a
Door	Door	connected
Room	Room	connected
Building	Building	connected
Building	Room	connected
Building	Door	connected
Room	Door	connected
Room	Building	connected
Door	Building	connected
Door	Room	connected

[0115] Defining the ontology projection on which the application operates is a step in the design, likely performed by the domain specialists, or with their significant help. For the complexity of enterprise applications, the definition of an ontology given here is simple. In practice sophisticated ontologies are typically used, and many more properties added to the specification. The definitions are kept simple to focus only on the properties required for the application specification.

[0116] Operators Algebra

[0117] The second dimension is a projection of the application onto the set of processing points the application uses. The processing points are defined as an algebra. Each operator of the algebra is defined as an atomic computational block with precise input and outputs, encapsulating a functionality and being implemented in a specified programming language.

[0118] The behavior of an operator is triggered by sending a message (called an event, carrying a data item called a payload)) on one of the inputs. There are two kinds of event stimuli that trigger an operator: process notifications, which are asynchronous input events, and compute requests, which are synchronous input events that require an immediate computation and return of a result.

[0119] During processing, an operator can send notifications to other operators: asynchronous send notifications); and/or request some synchronous processing from other operators via call requests). Hence, operators only declare and have knowledge of their own four kind of ports: (input/output and synchronous/asynchronous); process notification (asynchronous input); compute request (synchronous input); send notification (asynchronous output); and call request (synchronous output).

[0120] The implementation of operators concentrates only on the processing being done by that particular operator and is agnostic of where such external input stimuli are generated or where its output stimuli are actually sent. An application operator is not necessarily a pure software component. The application operator can equally be an analog computation device encapsulated within a digital interface (e.g. a thermometer).

[0121] There are two different kind of stimuli: explicit events sent by other operators, as described above; and encapsulated stimuli, like hardware signals (e.g. digital thermometer), operating system signals (e.g. clock), an external software trigger (e.g. a database trigger) or an external network call (e.g. REST call), that are encapsulated in the operator implementation and not visible in the operator specification.

[0122] The encapsulated stimuli are not visible in the specification of the operator, as they cannot participate in creating a normal dataflow of information while the application is being designed. Encapsulating them within a single operator enables the library designer to transform such complex and heterogeneous signals into a single form of event that the runtime can uniformly process.

[0123] Operators encompass a broad set of functionality, for example: common sensors or actuators from hardware libraries (e.g. camera, alarm); common machine-learning operations (e.g. face recognition) from software libraries; common time series analysis; domain specific

operators (e.g., machine-learning component for sneeze recognition); or common support and infrastructure operators (e.g., database interactions, logs).

[0124] Each operator has a set of parameters that control the details of the behavior of the operator. Such parameters are initialized before the operator instance code is executed and remain static for the duration of the operator. The set of operators available to an application together with their interfaces is the operator algebra.

[0125] The operator algebra consists of a set of operators O , where each operator in O has: a unique identifier name; a set of parameters P ; and a set of ports classified into four subsets. The subsets include input ports, which can be: asynchronous input ports, called the process notification ports RN; or synchronous input ports, called the compute request ports COR. The subsets include output ports, which can be: asynchronous output ports, called the send notification ports SN; or synchronous output ports, called the call request ports CAR.

[0126] Returning to the running example, to implement this application there is a pipeline that starts at cameras and ends at notifications on a monitor that is available to security personnel. Camera sensors generate images, such as binary data. A cropping operator crops faces from the image. The recognition operator compares a cropped face with a set of recognized faces and generates either (i) a recognized person, where a business key is added to the image, or (ii) an unrecognized person, where the input cropped image is pass unchanged along the pipeline. Recognized people are passed to an enrichment operator that uses the business key to lookup additional information about the person from an external database. A monitor displays information about known and unknown people via messages sent from the recognition operator and the enrichment operator.

[0127] From the running example, the application operators and their ports are: a heartbeat operator that sends a periodic message, without any particular input. The heartbeat operator ports include a send notification port heartbeat. A CameraPiSensor operator captures an image given an external stimuli, as follows a process notification port heartbeat; or a send notification port image. A Crop operator crops the image to the set of faces that appear, with the following ports including a process notification port image; and a send notification port *oneface*. A Recognize operator recognizes a face as a known or unknown person from a cropped image, as follows: a process notification port face; or two send notification ports known and unknown. An Enrich operator adds additional information about a known person, fetched from a database, as follows: a process notification port input; a send notification ports output; and a call port fetch. A Monitor operator displays the known and unknown people on a dashboard, as follows: a process notification port known; and a process notification port unknown.

[0128] The Heartbeat operator is customized by one parameter that controls the time interval of the heartbeat. This parameter can be either a constant numerical value, or a distribution that describes that random intervals. As discussed, most of the operators respond to explicit stimuli, except for the Heartbeat operator that generates a stimulus internally based on the operating system clock. The operators algebra is not a programming library in the traditional sense. Each operator can be implemented in its own programming language, yet they can still cooperate as part of an application, as long as they adhere to this simple, yet formal, event-based interface. Also, even if operators resemble classes, they are not classes in the traditional sense, as the interface is rigid compared to that of the set of methods of a class.

[0129] **Application Dataflow**

[0130] An application is specified as a dataflow graph that combines a chosen set of operators via information flow links. This is similar in spirit to a query execution plan, which is a dataflow combination of relational algebra operators.

[0131] There are two differences between dataflow and the application dataflow graph. First, unlike code blocks in such programming languages, operators in our algebra implement a relatively large functionality, not a simple programming statement. Examples can be a complex image analysis, a time series analysis, an automata with complex behavior, etc. Second, the core functionality as well as the inputs and outputs of operators are expressed in terms of the application domain, so they make sense for both a domain expert and for an engineer, and not only to the later, as it was the case in the past.

[0132] The context of an application defined as above is shared by many applications (or variations of applications) of the same vertical domain. The third application projection is the dataflow. The core of the application specification is the dataflow graph that describes processing and information flow.

[0133] The dataflow specification is now described. This graph of information is defined extensionally, as follows. The dataflow specification graph (DSG) includes : a set of tasks T as nodes; a function $operator : T \rightarrow O$, where $operator(t)$ is the operator in the application's context implementing the task t , where $t \in T$. The ports of the task t are defined by its operator $operator(t)$; a set of edges (called connections C) between pairs of nodes $(t1, t2)$, labeled with pairs of port names $(p1, p2)$, of the form $(t1, p1) \rightarrow (t2, p2)$, with the natural constraints: – $p1$ is an output port of $t1$ and $p2$ is an input port of $t2$, if $p1$ is a send notification port then $p2$ is a process notification port, and if $p1$ is a call request port then $p2$ is a compute request port.

[0134] There are two defined functions: $nodes : DSG \rightarrow T$ that is the set of nodes of a DSG; and $edges : DSG \rightarrow C$ that is the set of edges of a DSG. The running example has following task declarations, each with the associated operator: (heartbeat, Heartbeat); (camera, CameraPiSensor); (crop, Crop); (recognize, Recognize); (enrich, Enrich); and (monitor, Monitor). The connection declarations of the form $(from_task, from_port) \rightarrow (to_task, to_port)$ that "wire-together" the application include, as shown in graph 500 of the running example: $(heartbeat, heartbeat) \rightarrow (camera, heartbeat)$; $(camera, image) \rightarrow (crop, image)$; $(crop, one\ face) \rightarrow (recognize, face)$; $(recognize, known) \rightarrow (enrich, input)$; $(recognize, unknown) \rightarrow (monitor, unknown)$; and $(enrich, output) \rightarrow (monitor, known)$.

[0135] Anchoring the dataflow specification to entities is now described. Each node in the dataflow graph represents an abstract processing point, implemented by a particular operator. The dataflow graph does not include sufficient information for describing entirely the semantics of the application without an ontology. For example, the questions of "How many cameras are there? How will the "set of known people" be gathered? What are the known people per room or per building?" are purposely not answered by the dataflow alone.

[0136] In order to have a precise application semantics, additional information is obtained by adding a mapping between the dataflow graph and the application's ontology. Tasks are mapped in the dataflow to entities and dataflow links to relationships as a way to specify the communication relationship between particular instances of the ontology.

[0137] Informally, when a task is mapped to an entity, the data processing system creates one instance of that task per instance of that entity in the instance graph. Moreover, given a relationship that is labeling a certain anchored dataflow, messages sent on a runtime dataflow by corresponding entities will only be sent and received by targets that are linked via the corresponding relationship in the anchored dataflow.

[0138] Isolating the anchoring (mapping) from both the dataflow itself as well as the entity ontology contributes significantly to the conciseness of the specification, and to the separation of concerns.

[0139] Given a dataflow specification graph DSG , an anchor dataflow specification graph $ADSG$ includes the original graph adorned with two additional mappings: a function $scope : nodes(DSG) \rightarrow E$ mapping each task $t \in T$ into an entity in E ; and a function $relationship : edges(DSG) \rightarrow R$, labeling each edge between nodes $(t1, t2)$ in DSG with a relationship name r in the entity ontology, such that it exists in the ontology graph an edge $(scope(t1), scope(t2))$ labeled r .

[0140] Dataflows are graphs of operators layered on the application context, as shown in Table 2. Both graphs can be designed, extended, combined, and decomposed in ways that are understood by domain experts in addition to application engineers.

[0141] Table 2. The scope function maps tasks to entities - the values of the scope function for the running example.

Task	Entity
heartbeat	door
camera	door
crop	door
recognize	room
enrich	room
monitor	building

[0142] Application Instances

[0143] A particular application is the instantiation of the application specification described above to a particular instance of the entities ontology described as follows. An instance graph of an entities ontology E includes: A set of instance nodes I , each labeled with unique identifiers `instance_id`; a relation $IE \subseteq I \times E$, which declares for each pair $(i, e) \in IE$ that the instance i belongs to the set of instances of the entity e ; and a set of edges between pairs of nodes $(i1, i2)$, each labeled with a label $r \in R$, such that it exist two entities $e1, e2 \in E$ such that $(i1, e1), (i2, e2) \in IE$ and it exists in the ontology graph a relationship declaration between the $e1$ and $e2$ labeled r .

[0144] The relationship IE naturally defines several functions: $instances : E \rightarrow 2I$ defined as $instances(e) = \{i \mid (i, e) \in IE\}$; $entities : I \rightarrow 2E$ defined as $entities(i) = \{e \mid (i, e) \in IE\}$; and $reachable_instances : I \rightarrow 2I$ defined as $reachable_instances(i) = \{i1 \mid \text{there exists a path } i \rightarrow \dots \rightarrow i1 \text{ in the instance graph } I \text{ with the same label } r\}$.

[0145] The running example has six instances: one building, two rooms, and three doors, as shown in Table 3.

[0146] Table 3. The instantiation of the entity ontology for the running example.

Entity	Instance
building	building/1/
room	building/1/room/1/
room	building/1/room/2/
door	building/1/room/1/door/1/
door	building/1/room/2/door/1/
door	building/1/room/2/door/2/

[0147] The edges of the instance graph are shown in Table 4. The data processing system generates edges for the instances of the entities with *has_a* relationships. The complete set of edges in the instance graph is the transitive closure of this set of edges, the reverse edges of the transitive closure unioned with the identity relation. Fig. 6 shows the instance graph 600 of the running example with only the *has_a* relationship.

[0148] Table 4. The *has_a* edges of the instance graph for the running example.

Start node	End node	Relationship
building/1/room/1/	building/1/room/1/door/1/	has_a
building/1/room/2/	building/1/room/2/door/1/	has_a
building/1/room/2/	building/1/room/2/door/2/	has_a
building/1/	building/1/room/2/	has_a
building/1/	building/1/room/1/	has_a

[0149] The complete instance graph includes the *connected* relationship, calculated as the transitive closure of the *has_a* relationship, union the inverse of the transitive closure, union the identity relationship, resulting in the following instance graph shown in Table 5.

[0150] Table 5. The complete instance graph includes the transitive closure of the *has_a* edges as listed in this table.

Start node	End node	Relationship
building/1/	building/1/	connected
building/1/room/1/	building/1/room/1/	connected
building/1/room/2/	building/1/room/2/	connected
building/1/room/1/door/1/	building/1/room/1/door/1/	connected
building/1/room/2/door/1/	building/1/room/2/door/1/	connected
building/1/room/2/door/2/	building/1/room/2/door/2/	connected
building/1/room/1/	building/1/room/1/door/1/	connected
building/1/room/2/	building/1/room/2/door/1/	connected
building/1/room/2/	building/1/room/2/door/2/	connected
building/1/	building/1/room/1/	connected
building/1/	building/1/room/1/door/1/	connected
building/1/	building/1/room/2/	connected
building/1/	building/1/room/2/door/1/	connected
building/1/	building/1/room/2/door/2/	connected
building/1/room/1/door/1/	building/1/room/1/	connected
building/1/room/2/door/1/	building/1/room/2/	connected
building/1/room/2/door/2/	building/1/room/2/	connected
building/1/room/1/	building/1/	connected
building/1/room/1/door/1/	building/1/	connected
building/1/room/2/	building/1/	connected
building/1/room/2/door/1/	building/1/	connected
building/1/room/2/door/2/	building/1/	connected

[0151] The application specification and runtime configuration are now described. An application specification includes the following components: a context, composed of a real world entities ontology and an operator algebra; and an anchored dataflow specification graph *ADSG* in this particular context. A runtime configuration includes the following components: an application specification; and an instance graph corresponding to the real world ontology graph. Fig. 7 shows a graph 700 including relationships between the operators, ontology, dataflow, instances and the context, specification and runtime system.

[0152] Semantics of an Application

[0153] The natural semantics of the resulting application are described, via the extensional runtime graph. The runtime graph is the actual graph of operator instances that is being executed. The runtime graph, *RG*, corresponds to a runtime configuration composed of a triple (context, anchored dataflow specification graph *ADSG*, instance *I*). The runtime graph is defined by: a node labeled $t(i)$ for every task $t \in T$ and every instance $i \in I$ such that $entity(i) = scope(t)$; an edge $(t_1(i_1), t_2(i_2))$, labeled with $(port_1, port_2)$, for every edge (t_1, t_2) labeled $(port_1, port_2)$, in the dataflow specification graph, and for every pair of instances (i_1, i_2) such that i) $entity(i_1) = scope(t_1)$ and $entity(i_2) = scope(t_2)$, ii) there exists an edge $(i_1,$

i_2) labeled r in the instance graph, and iii) the *relationship* of the edge (t_1, t_2) in the anchoring of DSG is r .

[0154] In the running example, the runtime graph nodes are given by the instantiation, as shown in Table 3, and edges, as shown in Table 4, that define the runtime graph of the application. Fig. 8 shows a runtime graph 800 of the running example.

[0155] Default application semantics are now described. Given a runtime configuration, the semantics of the corresponding application on this particular instance is described by the operational execution of its respective runtime graph. Each task-instance $t(i)$, aka node of the runtime graph, is conceptually an actor-- the object contains a local thread of control, operates autonomously, and communicates via messages.

[0156] The execution is described by the following algorithm: For phase I, objects are initialized. For every task instance $t(i)$ in $nodes(RG)$, i) create the associated object as an instance of the operator O of the task t ; ii) add to this object all outgoing links according to the runtime graph RG ; and iii) initialize the parameters P according to the parameterization semantics. For phase II, for every task instance $t(i)$, the associated object loops forever: i) wait for a stimulus, i.e., a message arrives on a process or compute port; ii) if a message m arrives on a process port p , invoke the method $t(i).process_p(m)$ of the operator o of the task t , and no return result is expected; or iii) if a message m arrives on a compute port p , invoke the method $t(i).compute_p(m)$ of the operator o of the task t , and compose a response message and return it.

[0157] An event received on an input port triggers the invocation of an operator code, which may in turn trigger a hardware component. Typically, the operator performs some computation and then, as part of its logic, may send output events via send ports. To accomplish these actions, the operator code invokes $t(i).send_p(m)$ to send a notification via port p . Such events will be asynchronously transmitted to all the task-instances that are connected to that send notification port in the runtime graph.

[0158] In order to implement the operator semantics, the operator might also send events on call request ports. To accomplish this, the code invokes $t(i).call_p(m)$ to call via port p . The semantics of a call are to block the actor, wait for a response, and then process the result synchronously. Because of the structure of the runtime graph, multiple task-instances may respond to a call. The operator takes the first response and discard the remaining responses. If no response arrives within a timeout, a runtime error is generated, which can be caught and handled by the given operator.

[0159] By definition, when a *send ()* or *call ()* is executed, the message contents are immutable. This requirement isolates the message contents from any side-effects on the message from connected upstream or downstream task-instances, greatly simplifying debugging of the application.

[0160] The execution of the methods related to a task-instance is agnostic to the actual topology of the runtime graph. The code specifies only sending a message on a certain port, but the specific instances that are the recipients of the message are not known at code-writing time. The recipients of the message are determined by the links in the instance graph plus the connections between tasks in the application specification according to the definition of the runtime graph. The runtime system directs the message towards the right recipients.

[0161] In the running example, when the task-instance (crop, building/1/room/1/door/1/) sends a message to its output port *oneface*, the message is broadcast to all the instances connected via the runtime graph. In this case, the single task-instance (recognize, build/1/room/1/). In other words, the set of instances to process a message is derived from the combination of the dataflow graph and the instances. To add a message from the recognize task to its 'children' would simply require a send port for the *Recognize* operator, a process port for the *Crop* operator, and a connection between the two. The runtime system automatically manages all broadcasts that result from whatever instances are currently configured.

[0162] During the execution of the system, a task-instance can update its internal state (if any) during the computation of its associated operator. The internal state, available in local memory, is initialized via parameters or directly in the code. This internal state persists for the lifetime of the process executing the task-instance. Given that an operator (via a task) interacts with the external world only through messages (received or sent), this internal state is not directly visible to outside world. Despite the actor-based semantics, operators do not need to be implemented via an actor programming language.

[0163] The application semantics imposes rules on the execution of task-instance objects. This is called the guarantees of the semantics. First, there are no shared states between task-instance objects. This requirement reduces the number of implementation choices available to the library engineer. Operator implementations share state through notifications or calls. Second, messages and return values are immutable. This requirement simplifies the computational model that application engineers developing code must understand. Each message stimulus is thus executed in isolation from other message stimulus. Isolation semantics simplifies the computational model of tasks that the application engineer must understand. The

constraints reduce the engineering effort for the implementation of an operator, as certain implementation details (e.g. threads, locks) will never be visible, nor even available to the application engineer.

[0164] The semantics intentionally do not guarantee several other properties, including (1) no guarantees of durability for task instances; (2) no guarantee of message arrival and/or the number of time the message arrives (guarantee message durability); (3) No guarantee of message order execution; and (4) No guarantee of side-effect atomicity. Similar to durability, the application semantics do not require all-or-nothing atomicity for the set of messages sent by the invocation of a method. Extensions to the semantics that address these cases are subsequently described.

[0165] Generally, the application specification model is configured to balance simplicity, coverage, and resource consumption. Designing the right application specification model is the crucial point on which the various advantages can be obtained. In designing a model for applications there is a clear compromise to be made between: the simplicity of the model (how many people can understand the model); the coverage of the model (how many applications can be modeled in this way); and the implementation cost of the model (how expensive it is with respect to how many resources, money, computation, performance, etc., are required to implement the given model).

[0166] If the model is simple, it works well for a large audience as "application designers"; however, the set of the resulting applications is relatively small, and therefore not usable in large enterprise settings. At the opposite extreme, having a complete model that covers 100% of the world of applications can be prohibitively complicated for the average developer to use, let alone for non-developers application designers. Orthogonally, the model may include features that require too many resources (money or computational) to be implemented, and that is also a dimension that needs to be factored in while designing a model.

[0167] The application specification model described herein is configured to be usable and understandable by a large set of people. However, the simplicity derives from that fact that there is specified an application based on projecting in isolation each dimension of the application instead of the whole at once. Each such dimension is simple; the resulting applications, via integration of the projections, are complex.

[0168] Extensions of the model are described. In the described application specification model, the semantics of the runtime graphs does not require certain features that might be necessary for some applications: the order of messages, the atomicity of side effects of messages or durability of tasks or messages. Imposing such conditions by default (especially

for distributed execution) would impose a penalty. However, the need for such constraints may arise in some applications. Extensions of the model can include that messages sent from a specific task-instance to another can be guaranteed to be executed by the target object (a) exactly once or (b) at least once. This requirement imposes durability of messages for specific connections.

[0169] Changes to the internal state of specific task-instance must not be lost. This would impose the durability of task instances for specific tasks in the specification. All outgoing messages generated by specific instance tasks are sent, or none. This would impose the atomicity of side-effects for some specific tasks of the specification. An outgoing (set of) message communicates with the connected task-instance in a transaction, guaranteeing all-or-nothing and exactly once delivery of the message.

[0170] Each such additional constraint is similar in nature to service level application requirements (SLAs). They should be explicitly declared as a model extension and enforced by an extension of the existing runtime system.

[0171] Code isolation is now described. The application specification model includes an architecture where the code of each operator is completely agnostic to the rest of the application runtime: independent of the ontology, of the dataflow itself and obviously independent of the instance on which the application is applied to. Finally, each operator code is independent of each other.

[0172] This design simplifies and constrains the semantics of operators as much as possible to provide clear and readily understandable semantics for the code, without limiting the application engineer's expressive power in coding the operator, in most cases.

[0173] The engineer implementing an operator uses the local knowledge available at the operator level: the available call ports and send notification ports of the given operator; the send and compute ports; and the internal task-instance state of the instance.

[0174] Knowledge of other aspects of the system are intentionally not available during the application engineer's development of an operator, for example: the topology of the task connections in the specification; the anchoring of the application specification to the context; the topology of the instance graph; and any knowledge about the distributed communication structure of the system. Prohibiting access to this information means that a configuration of the system can be changed along these dimensions without requiring any modifications to operator code.

[0175] The application data model is configured for standardization and reuse. Because the data processing system projects the definition of an application on various dimensions and

creates the resulting application by a mathematical integration of such projections, there is an opportunity for standardization and reuse. Each individual projection may be usable in multiple applications.

[0176] Many applications are designed in the same vertical domain. Such description of the domain, materialized as the ontology in our model, is the base of application specifications. Such ontologies can be standardized for further reuse. Applications in a particular vertical, such as within a company, or across companies, can share the same entity ontologies (e.g. building management).

[0177] The operators code independence of both the entities ontology and the dataflow, as well as the particular instance the application applies to, makes it possible for such operators to be standardized and made available via libraries. For example, in the running example, both operators Heartbeat and Enrich are reusable operators that are likely useful in most applications, not only in this particular building management application.

[0178] Operators can be classified by two different criteria. First classification criteria is the generality of the operator. Some of them are domain specific (e.g. image analysis for a security application), while some others will be generic and usable across several different vertical domains (e.g. Heartbeat).

[0179] A second criteria is the level of operator specialization. In some cases the specialization of an operator is provided by parameters, e.g. in the case of Heartbeat. In other cases, one can standardize only operator templates, such as operators whose skeleton semantics and behavior is clearly defined but whose details needs to be specialized via code, not via simple values. For example, the skeleton code of an automata operator template is unique and standardized, independently of whether the automata simulates a pump or a door. The running example application includes many such operator templates: data enrichment, automata, logs, database interactions, windowing-streaming operators, and so forth.

[0180] Operators are naturally clustered into similar groups, forming libraries of operators. All those kinds of libraries, generic or domain specific, templates libraries or final libraries, should be subject to standardization to increase reuse and further minimize the new lines of code required to complete a new application.

[0181] In addition to ontologies and operators, another potential opportunity for reuse and standardization are fragments of dataflow. Several tasks in a dataflow, together with their links, can implement a certain greater granularity logical operation. Such logical operation may be useful to other applications, in the same vertical domain, or other vertical domains. For example, the subgraph of the dataflow graph in our running example that only achieves face

recognition, from the heartbeat, camera, followed by the crop of faces, and then all the way to the resulting recognized person, is certainly usable in more than one application. Such fragments can be a subject of standardization in order to further decrease the effort required for a new application.

[0182] Applications evolve over time, and very often the cost of evolving an application is the major cost factor in the total cost of ownership of an application, not the initial cost of building the first version. Given that an application is defined as a mathematical integration of a set of projections specified independently, every possible change of an application implies a certain modification across one (or more) of the various projections (e.g. the instance changed, or the dataflow has been modified, or the operator implementing a certain task has been changed, or simply the parameters of a certain operator changed). No other changes of the application are even possible because the application only exists as a virtual integration of such projections. This has two major advantages. First, the evolution is much simpler to understand, conceptualize and express. Second, the mathematical definition of the application allows to formally detect the impact of such deltas on the resulting application.

[0183] Many applications are built not from scratch but by customizing an existing application. By specifying the application via its various projections, the specification isolates the precise points in which an application can be customized, by simply asking a simple question: "on which dimension is my application different from this existing application?". The answer can only one of the possible choices: the parameters of some operators are different (e.g. images are taken by a camera every 15 seconds instead of 5 seconds), or the operator of a task can be different (e.g. use a database instead of a file system for a log task), the dataflow is slightly different, or the instance in which this application is based on is different. Such differences are simpler to conceptualize and specify in our model rather than customize a large code-base with intertwined functionality, and the impact of the changes much easier to follow.

[0184] Application specification (creating the design requirements) and application implementation have been traditionally parallel efforts. This model allows and forces the design to be integral part of achieving a running application. By requiring the design to be part of the final implementation effort, and by splitting the application design in various dimensions (ontology, data, code, etc.), a larger user base can be active participants in the application's final behavior. Some stakeholders, who do not necessarily possess coding skills, are allowed to specify part of the application design, on which the application behavior is actually based.

[0185] Table 6 shows a potential implication of various categories of stake holders in the design of applications.

Role	Entities	Algebra	Dataflow	Instances
Domain Specialist	Design	Design	Design	
Financial			Approve	
Software Engineer	Design	Design & Code	Design	
Usability Engineer			Design	
Hardware Engineer		Design & Code		
Infrastructure Engineer				Design
Operations IT			Analyze	

[0186] Table 6 shows a cross-reference between the roles of an application and the application specification. A design entry means most of the participation is in the design phase. A code entry means software engineering of code. An analyze entry means most of the participation is in analysis of the existing system.

[0187] Given a model as described, many efforts necessary for the implementation of applications can be streamlined. Some functionalities that can be semi-automatically added include i) designing user interfaces targeted for non-developers to specify ontologies and dataflows; ii) generating a complete application only from the application model, with zero additional lines of code, by providing a standard runtime; iii) expressing a distributed execution plan as an additional layer of information in top of the basic application model; iv) given an execution plan, calculating the (real) cost of an application for business purposes; v) automatically obtaining an "optimal" execution plan that satisfies the domain constraints and minimizes a target cost function; vi) given an optimization plan, obtaining a distributed execution automatically, with zero additional code; performing various analysis on the model (e.g., failure analysis, information leakage); vii) specifying security requirements as an additional modeling layer in top of the basic application model; viii) automatically modifying a model to facilitate testing; either for performance analysis purposes or for focused zoom-testing purposes; ix) automatically modifying a model for simulation purposes; x) computing incremental updates in case of changes in the underlying specification, and detect the part of the application that needs to change and how it needs to change; and xi) adding additional functionality as additional modeling layers over the basic model via functionality injection. Examples of such additional layers of functionality include error management logic, synchronization logic, logging logic, calibration logic, control logic, and user visualization logic. Such applications can be built as result of machine-learning based tools.

[0188] The Autocoder

[0189] The autocoder includes a data processing system that includes a runtime for the application specification model previously described. Fig. 9 shows a tool chain 900 from

application specification to execution. Arrows indicate the flow of information and code across the tools. The call-outs indicate the information provided or generated by a tool. The autocoder includes ten main tools that coordinate to design, implement, and execute an application.

[0190] A design tool provides domain experts with a visual representation of the specification and allows domain experts and application engineers to explore different design configurations. The output of the design tool is the entities, dataflow, and operator specifications, expressed in the Autocoder format.

[0191] A dataflow compiler tool accepts as input the specification files from the design tool. The tool verifies the specification's correctness and injects additional system functionality by rewriting various parts of the specification, but not the actual code. This tool is accessible to application engineers and produces output ready for the build scripts.

[0192] A code generation tool takes the operators algebra and the optimized local execution plan and automatically generates, for each operator, integration code and scaffold code. Integration code provides the bridge between the application code and the system runtime. Scaffold code is an outline of the application operator code, complete with required imports and stubbed method calls.

[0193] An IDE tool (implemented as plugins to a favorite existing tool) to provide actual application code for the stubbed methods produced by the code generation tool. Users can use additional tools to implement sensors and actuator code (e.g., tools that flash code to devices for testing).

[0194] An instance identifier tool applies the instance identifier models to dataflow tool output to generate an extended dataflow.

[0195] A customization tool allows engineers and business users alike to fine tune the behavior of various parts of the application by assigning values to the parameters of operators in the application.

[0196] An optimization tool searches the space of possible variations of the specification in search of an optimal execution plan according to an objective function. This search can be done completely automatically or as part of a design loop with an optimization engineer. The output of the optimization tool are two execution plans. The local execution plan focuses on debugging in a simple centralized environment.

[0197] A distributed execution plan focuses on more complex testing, pre-production and production releases.

[0198] A build tool takes the extended dataflow, the optimized distributed execution plan and outputs a set of build scripts for the selected external build tools. The build scripts take the

same information as input, plus the system runtime code, the parameters file and the application code, and outputs a collection of containers. Each container includes the Autocoder runtime and system libraries, the subset of the application specification relevant to the container, the relevant subset of the application code, and the relevant libraries for the code.

[0199] An install tool takes the dataflow tool output, the optimized distributed execution plan, the specification of the actual instances, and outputs a set of install scripts for the selected external install tools. The scripts take as input the additional information about the host target CPUs and install the containers on the target machines.

[0200] An execution tool initializes, starts, and controls the execution of the containers on the target machines. During the initialization of the container execution, the runtime graph is first constructed. Starting a container will trigger the start of all task-instances in that container. The execution tool also controls pausing and shutdown of containers and managers errors happening at runtime. The systems operations engineer monitors operations and uses the execution tool to modify the state of containers.

[0201] Autocoder also has an additional suite of tools for extensions, such as computing optimal execution plan for a specification, exploring what-if analysis of failures to the application, functionality injection, optimization of the application parameters according to given metric, and incremental changes the runtime system.

[0202] In Autocoder, the default language for operator code, system runtime and the various tools is Python. application operator code can be written in other languages. Every additional programming language used for operator implementation involves a port of the runtime and system libraries and the rewrite of the code generator tool, but not the other Autocoder tools. In the end, the application can be a mix-and-match of operators implemented in various programming languages, and the Autocoder tools support this case.

[0203] An application specification syntax is now described. The Autocoder receives a description of components of the application specification including: a description of the entities ontology E (e.g., a list of entities and a description of the relationships between them); a description of the operators algebra O (e.g., a list of available operators, each with its own ports); an anchored dataflow specification graph $ADSG$, which adds scope labels to task nodes in the dataflow, and relationship labels to edges in the dataflow; and the instance graph. Each such part can be described as an input to Autocoder by a YAML formatted file. Fig. 10 shows a partial view 1000 of the entities file for the running example and is missing the declaration of the "connected" relationship.

[0204] The syntax of the entities file is now described. The entities file contains a description of a specific entities ontology, such that of Fig. 10. The file lists the entities of the system and the relationships between the entities. The file format is a direct representation of the ontology.

[0205] The syntax of the operators file is now described. The operators file, shown in Fig. 11, includes a description of a specific operator ontology, with additional information. The file lists the operators and their ports. If a type of port is missing for an operator, then the system assumes no ports of that type.

[0206] In order to execute an operator, Autocoder also requires additional information about the implementation specified as sub-properties of the implementation property of the operator: the implementation language (language), the location of the code (module), and the necessary libraries for building the operator (pip in the case of Python).

[0207] Fig. 11 shows an example view 1100 of specification and implementation details of the CameraSensor operator in the operators file for the running example. The input ports require corresponding code to implement them. The system generates code for output ports that communicate with other tasks.

[0208] The syntax of the anchored dataflow file is now described. The dataflow file 1200 of Fig. 12 includes a combined description of the dataflow graph and the anchored dataflow graph. Fig. 12 shows the tasks and connections for the running example.

[0209] A syntax of the instances file is now described. The instances file contains the description of the instances graph on which the application operates, as shown in view 1300 of Fig. 13). The file lists the instances of the entities ontology on which the application operates, and the relationships between the instances. The file format is a direct representation of the definition of an instance.

[0210] Instances represent virtual and physical real world objects. Instances can have unique instance identifiers (strings in this case) that are used to coordinate various functionality across the system. In the running example, the instance graph reflects a static world and are provided via a file. For simplicity of this section, assume that the instance graph is available to Autocoder, and that it does not change during dataflow compilation and execution. In practice, applications have a dynamic set of instances provided by an external system (e.g. a database). Furthermore, only part of the compilation described in this paper depends on the existence of the instance graph; most of the compilation depends only on the application specification itself and is agnostic to the particular instance on which the application operates.

[0211] **Fundamentals Of Autocoder**

[0212] A centralized execution scenario where the entire runtime graph executes in the same process is now described. The additional steps that are needed for a distributed execution are described, where parts of the runtime graph are distributed to multiple processes and heterogeneous hardware.

[0213] In Autocoder, a running application executes the semantics by applying the standard actor execution strategy to each task-instance node in its runtime graph. Task-instances are objects of particular classes that combine application code with system code. Each task-instance labeled $(task, instance_id)$ is an object of a class OP , where OP is the Python class associated with the operator of the respective $task$. For each operator op in the algebra that declares its implementation via the OP class, a class OP exists in the runtime. Operator classes mix, through inheritance, system code with application code. During the execution of an operator, messages are exchanged between task-instance objects.

[0214] Standard Single Process Application Execution

[0215] Task-instances communicate with the external world via messages. When sending a message or processing a message, the data associated with that message is passed as the payload to the Event class. Objects of this class keep two data items: (a) the payload that holds the content of the message and (b) the provenance that holds a sequence of provenance stamps. Each stamp describes a point in the of history of the message's production.

[0216] Autocoder distinguishes two kinds of messages: system messages and application messages. Application messages are the normal messages generated by applications. System messages are generated by the system and are used for implementing functionality such as error management, control, status reporting, etc. System messages are invisible to the application.

[0217] For actor semantics, the following overall design decisions form the basis of the implementation. A task-instance has a unique thread and all code of the object is processed by this particular thread. Task-instances react upon receiving a message on an input port by invoking the process(message) method. Processing of process(message) is protected by a lock of the task-instance. Only one message at a time is being processed in a task-instance.

[0218] Arriving messages are placed in a FIFO queue. The thread of the task-instance call process(message) on each message in the queue, in the order of arrival. A variation to the FIFO ordering is subsequently described. Also, several optimizations subsequently described result in changing the order of messages processing.

[0219] Processing a process(message) has two parts: (a) the system processing part that deals with ensuring standard semantics (e.g. isolation through locks) and (b) the application code that implements the logic of the port. The application logic is wrapped with system logic.

[0220] The system part of message processing implements additional standard functionality such as error management, provenance management and type checking of message payloads.

[0221] As part of application logic, new synchronous or asynchronous messages can be sent to other task-instances. The application uses call(data) to for synchronous messages and send(message) for asynchronous messages.

[0222] Processing of a send(message) is composed of two parts: (a) system code that deals with provenance management and (b) dispatch of the message to all the targeted task-instances, on their respective input ports, according to the topology of the runtime graph. Processing of call(data) is described subsequently.

[0223] Incoming messages are deep copied upon receipt or are part of a compute to avoid any kind of state sharing between task-instances.

[0224] Autocoder implements its system code and imposes a certain structure on the application code in order to achieve the logic described above. The result is a generic class structure that organizes the system and application code into an object-oriented class inheritance hierarchy, as shown in structure 1400 of Figure 14.

[0225] System and application operators are provided with a collection of shared functionality. This functionality is implemented in the GenericTaskInstance class, which is the root of the operator classes hierarchy.

[0226] Autocoder offers different variations of application execution semantics. These optimized variations are coded as system subclasses of GenericTaskInstance. Currently Autocoder provides the standard actor semantics in the TaskInstance class, and a variation of actor semantics that uses less resources in the PassiveTaskInstance class, as described in Section 4.2.1. Several further optimizations are implemented via parameterization of those two basic system classes.

[0227] For every send port *s* or process port *p* defined in the operator specification, the application code corresponding to that operator will have the corresponding two methods: send_s() is system implemented but available for use in the application logic; or process_p() that includes the application engineer's implementation of processing messages that arrive on port *p*. This method is invoked under cover of the system version of the process() method.

[0228] The Code Generator tool of Autocoder automatically generates, for each operator, Python code, in order to impose the required structure. For an operator *op* two classes are generated, a class *Auto_Op* and a class *Op*. The class *Auto_Op* inherits from the system classes that implement the target semantics for the operator (standard actor or optimized versions thereof). Each automatically generated stub class *Op* inherits from class *Auto_Op* and includes the scaffold of the operator implementation itself. This class is to be implemented by the application developer.

[0229] Fig. 14 shows a class inheritance hierarchy 1400 for operator code. An arrow indicates inheritance. Operator *Op1* has the standard task-instance actor semantics. Operator *Op2* has passive (thread-less) semantics. The inheritance of application classes from system classes enables the implementation of message processing via wrapping system calls around application code. To communicate, task-instances, which are instances of application classes *Op*, send messages to each other through ports.

[0230] The application code is organized so that message sending and receiving is done through system code. When the code of an operator *Op* calls a *send_p()* method to send a message on port *p*, the call is forwarded to the system implementation of the *send()* method, as shown in Fig. 15. This method routes the messages to the destination task-instances, all of which have a system *process()* method. The system *process()* method then dispatches the message by calling the appropriate application method in the task-instance code.

[0231] Fig. 15 shows a process 1500 for processing a *send* and a *process*. An arrow indicates the direction of data flow. The send call executed in an application operator task-instance is forwarded to the system implementation of send. This method sends the message to one or more task-instances where system implementation of the process method is invoked. The process method then dispatches the message to the application operator process method.

[0232] A *GenericTaskInstance* class contains functionality that applies across system classes and application operators. Objects of this class have several kinds of data fields. First, a runtime graph information, composed of: i) *task_name* of type *STRING* that stores the name of the task of this task- instance, ii) an *instance_id* of type *STRING* that stores the instance identifier of this task- instance, and iii) *connections* of type *MAP* that stores a dictionary $output_port \rightarrow \{(task_instance, input_port)\}$. The task-instance's outgoing edges in the runtime graph are indexed by outgoing port. Note that edges in the runtime graph link output ports of task-instances to input ports of task-instances. Links to task-instances are direct Python object references, and do not go through any other indirection method. In general, these fields are not accessible to the application code. Exceptions are marked below. A second data field

includes system fields for normal processing of a task-instance: i) system parameters of type MAP that stores a dictionary *parameter* *__name* $\rightarrow$ *value*. System parameters control the execution of a task-instance and are not accessible to application code. The lifetime and the role of task-instance system parameters will be discussed subsequently; ii) system operational data stores the system fields that are required at runtime for execution. Examples include the lock that is protecting message processing, and information about the runtime state of the task-instance. In general, these fields are also not accessible to the application code. Exceptions are detailed bellow.

[0233] A third data field includes application fields required by the application, and obviously available to the application. Examples are the application parameters of type MAP stores a dictionary *parameter* *__name* $\rightarrow$ *value*.

[0234] The GenericTaskInstance class contains two primary methods. The process() method processes incoming messages. The send() method processes outgoing messages. Call and compute methods are described subsequently. The process() method has the signature process: PORT_NAME $\times$ EVENT $\rightarrow$ None. This method is the external entry point to the GenericTaskInstance class for messages to be processed. The method first invokes system code to ensure: message isolation, automatic type checking (if necessary), error management, provenance management, and a deep copy on incoming messages events in order to ensure task-instance isolation. The method starts by executing the system code that implements the functionalities listed above. The code eventually invokes an internal generic method process_internal() for message processing, which calls the application operator implementation of the process() method for that particular input port. At the end of the application logic, the system method continues to ensure the functionalities above. In this sense the application code is wrapped by system code.

[0235] A simplified sketch of the methods is as follows. Error management, type checking, and message copying and provenance methods are skipped for simplicity:

```
def process(self, input_port, event):
    self.process_internal(input_port, event)
    def process_internal(self, input_port, event):
        with self.lock:
            instance_state = call_instance_state_read(self.instance_id)
            fun = getattr(self, 'process_' + input_port)
            fun(event)
            call_instance_state_write(Event(self.instance_id, instance_state)
```

[0236] The indirection from `process()` to `process_internal()` gives the subclasses an opportunity to override and change behavior at this point. The system locks the task-instance object when the external `process()` method is invoked. The lock is released when the method finishes. However, potentially, deadlock is a consequence of this locking arrangement if there is a cycle in the dataflow specification, and potentially in the runtime graph. The design tool reports a warning when a cycle is detected in either of the above.

[0237] The `send()` method has the signature `send: PORT_NAME  $\times$  EVENT  $\rightarrow$  None`. For each task-instance connected to the output port in the runtime graph, invoke the task-instance `process` method on the corresponding input port, passing the event as argument. This direct object function invocation does not go through any other indirection or system code. The sketch of the code is as follows.

```
def send(self, output_port, event):
    for conn in self.connection[output_port]:
        conn.task.process(conn.port, event)
```

[0238] The `TaskInstance` class inherits from the class `GenericTaskInstance` and implements the standard actor semantics described previously. In order to achieve this goal, the class has additional fields: *thread* is a thread object whose live execution triggers the method `active_run`, *queue* is the incoming FIFO queue of events to be processed, and additional state and execution flags used to coordinate the thread and its execution.

[0239] The `TaskInstance` class has a set of methods that implement the semantics of message processing: `active_run()`: The signature of the method is `active_run: None  $\rightarrow$  None`. This method is the main method of the internal thread of the task-instance. The method performs an infinite loop; at every iteration of the loop the method `run_once_maybe()` is invoked; `run_once_maybe()`: The signature of the method is `run_once_maybe: None  $\rightarrow$  None`. In a normal scenario this method simply reads an event from the queue (under lock), and invokes the normal processing `process_internal()` described in the superclass `GenericTaskInstance` on that event. This method might do nothing if no event is present in the queue. A sketch of the code for `run_once_maybe()` is as follows:

```
def run_once_maybe(self):
    try:
        (input_port, event) = self.queue.get(timeout=0.5)
        self.process_internal(input_port, eventset)
    except queue.Empty:
        pass
```

[0240] `process()`: The method has the signature `process: PORT × EVENT → None`. This method overrides the superclass `process()` method and becomes the external system entry point to the `TaskInstance` class for messages to be processed. Instead of dealing immediately with the event as its superclass does, it simply queues it for later processing. When the internal thread is free, it will invoke the normal superclass `process_internal()` method as part of the `run_once_maybe()`, and processing will continue as described before.

[0241] A sketch of the process method is as follows.

[0242] `def process(self, input_port, event):`

[0243] `self.queue.put((input_port, event))`

[0244] The queue is lock protected because of potential multiple threads accessing it (multiple writers and a reader). It might appear that the generic task-instance lock taken in the method `process()` is unnecessary. However, given the optimizations subsequently described, the lock is still necessary in the generic method, and avoided when not necessary.

[0245] While writing the application code (aka non Autocoder code), there are multiple levels of external developers that are distinguished, classified based on the degree of knowledge they have about the system details and how much do they use in their code. First, pure application developers do not have knowledge of any system details, threads, queues, locks, and so forth, and their code is completely agnostic to the system details. Second, library developers implement, for example, operators that correspond to sensors. Very likely such code will wait on some software/hardware signal and then react. This application code has a detailed knowledge of the system implementation of the class `TaskInstance`. Typically such code will overwrite the method `run_once_maybe()` or even the method `active_run()`. Such code should understand the internals of the class `TaskInstance`: queues, threads, state fields and locks. Library developers are a special class of developers who have more in-depth knowledge of the system.

[0246] Every operator that overwrites one of the methods `run_once_maybe()` or `active_run()` has to be signaled to the Autocoder system. Several optimizations (e.g. multithreading, multiplexing) will need to be analyzed with care for those special operators. Developers will use additional syntax to signal to Autocoder that the given operator uses internal system knowledge, particularly threading. The new syntax is part of the implementation section of the operator specification. Such operators are declared as being active.

[0247] For example, the specification of a *Heartbeat* operator 1600 of Fig. 16 declares an active execution strategy, as it needs to manage its local thread and interact with the system

clock. In a case when the operator makes no use of system threads, the operator uses a passive execution strategy. This strategy is explicitly declared in the operators specification, or inserted automatically as a default value as passive if the strategy field is not present.

[0248] Operators can be without explicit external stimulus. In some cases, an operator has no external stimuli and thus no process or compute ports but are triggered by external software and/or hardware events (e.g. clocks, sensors). Such operators need to have an internal thread of operation, and hence need to be marked as active.

[0249] Sending external messages to other connected task-instances via the `send()` and `call()` methods are available to the engineer in the implementation of `run_once_maybe()`. In addition, the implementation must somehow pause or delay or wait on external software/hardware, otherwise the system will attempt to process an infinite loop of calls to `run_once_maybe()` as fast as possible.

[0250] Take as an example the code of the heartbeat operator 1600, which sends an outgoing notification after delay seconds, with input from the system clock, not any external task-instance. The implementation of the heartbeat operator will overwrite `run_once_maybe()`. This method is continuously called by the runtime system via the `active_run()` method, repeatedly giving control of the system thread to the implementation of the operator. In this case there are no events to be read from queue, but an event has been constructed around the operating system call that delays the thread.

```
def _run_once_maybe(self):
    self.event.wait(self.delay)
    self.send_heartbeat([self.event_factory()])
```

[0251] Fig. 16 shows a simplified version 1602 of the heartbeat operator 1600. The amount of delay is given as a parameter. The task-instance then has the thread wait for the delay. After the delay, an empty message is sent on the heartbeat port. The method ends and is immediately called again as defined by the system implementation.

[0252] According to the standard semantics, processing a message on a certain port must guarantee message isolation. However, in certain cases there is deviation from the standard semantics. A *process* port may be declared fast. Fast ports are special ports where: the queue is not utilized,

[0253] when executed, locks are not taken, and the system and application logic execute on the external calling thread, not the task-instance object internal thread.

[0254] The system implementation uses fast ports with system messages for control and synchronous calls.

[0255] Management of a task-instance. Every task-instance object goes through several processing states thought its lifetime, as follows: Initialized: Upon object creation and initialization, the necessary fields of the object are created and the task-instance is in the Initialized processing state, yet the task-instance is not yet operational from the dataflow point of view. Running: Upon starting, the task-instance is ready to receive and process events in an infinite loop. The task-instance is in the Running processing state. Paused: The loop can be temporarily stopped by transitioning the task-instance to the Paused processing state. While in this state the incoming messages continue to be enqueued but not processed, and Finished: the task-instance transitions to Finished processing state

[0256] This four state automata controls the execution of every task-instance. Obviously, the execution state of a task-instance is opaque to the application code. The automata graph 1700 shown in Fig. 17 shows that the automata, and the task-instance, is in the Initialized and Finished activity states exactly once. The automata alternates between the Running and Paused states. While in either state, the task-instance can be stopped. In general, when the automata changes state via receiving a system event, a corresponding method is executed to adjust the activity of the task-instance by adjusting the corresponding internal fields.

[0257] The implementation of processing states depends on the threading design of the application semantics. An example is the behavior of the standard TaskInstance class, which has an internal thread processing the received messages. Processing state management is achieved with the help of the following flags: alive of type Boolean is only true when the task-instance is in either Running or Paused processing state; or running of type Boolean is only true when the task-instance is in the Running processing state (in case the object is already alive).

[0258] For the TaskInstance class the implementation of the previous four methods for processing state transitions is as follows. During object initialization, a threading object is created, and the variables alive and running are set to False. In addition, a synchronization object called restart is created. Threads can block and wake up based on this synchronization object.

[0259] The process_start() method sets the running status variable to True and starts the thread. The newly started thread starts executing the active_run() method. A sketch of the method is as follows:

```
def process_start(self, event):
    self.running = True
    self.thread.start()
```

[0260] The process_pause() method sets a flag indicating that the thread is no longer running. A sketch of the method is as follows. When the task- instance thread checks this variable value and finds it is false, the thread will block on the restart synchronization object.

```
def process_pause(self, event):
    self.running = False
```

[0261] The process_restart() method restart is accomplished by setting the running flag to True and triggering the restart synchronization object that the paused thread is waiting on. A sketch of the method is as follows.

```
def process_restart(self, event):
    self.running = True
    self.restart.set()
```

[0262] The process_stop() method first sets alive to False indicating the task-instance is declared not alive anymore. The method then triggers the restart synchronization barrier in the case that the task-instance thread is blocked in the Paused state. The controlling thread (the thread that called process_stop) waits for the task-instance thread to finish. This join may fail because (i) there is a race condition and the task-instance thread terminated before the controlling thread, or (ii) there is no running task-instance thread. A sketch of the method is as follows:

```
def process_stop(self, event):
    self.alive = False

    self.restart.set() # so paused thread also
    dies try:

        self.thread.j
    oin() except
    Exception:

        pass
```

[0263] Active_run uses additional state management variables is as follows.

```
def active_run(self):
    while self.alive:
        if self.running:
            self._run_once_maybe()
        else:
            self.restart.wait()
```



```
self.restart.clear()
```

[0264] Thread management is cooperative. The processing state transition methods just set some internal flags of the class, they do not directly affect the thread. The thread will change its processing based on the value of those flags. Updating those flags does not affect normal processing of messages, i.e. the normal process() methods are not interrupted. The state transitions methods are not executed on the internal thread of TaskInstance class but by the external calling thread. The mechanism by which those methods are invoked uses normal event processing, but uses fast ports described above.

[0265] Targeting an Instance in Send Calls. The default application semantics specifies that a send notification is broadcast to all the connected instances in the runtime graph. In some cases, a task-instance needs point-to-point communication with exactly one other known instance, instead of broadcasting to a set of instances.

[0266] Autocoder provides point-to-point communication in the form of task-instance to one other (single) task-instance, expressed in code via an additional argument to the send() method. The logic of the send() remains unchanged. The logic of the receiving task-instance discards messages received but not intended for a given task-instance. A sketch of the revised send() method is as follows. An additional target_instance_id argument added to the send() method. The type of the argument is an instance identifier. The argument is simply passed along to the process method.

```
def send(self, output_port, event, target_instance_id=None):
    for conn in self.connection[output_port]:
        conn.task.process(conn.port, event, target_instance_id)
```

[0267] The revised process() method simply discards any targeted sends that are not intended for its task-instance by comparing its customization to the provided target argument.

```
def process(self, input_port, event, target_instance_id):
    if target_instance_id is None or target_instance_id == self.instance.instance_id:
        self.process_internal(input_port, event)
```

[0268] The use of a targeted variation of the send method is problematic because it introduces a dependency that breaks the orthogonality between instances and tasks. The task-instance code will only function correctly when a particular instance identifier is part of the configuration, or when a correct task-instance is passed as data to an incoming message. The system runtime uses targeted sends to (i) control individual task-instances and (ii) for the implementation of call messages.

[0269] Implementation of Call and Compute. The application specification provides two separate means of communication between tasks: send()/process() notifications and call()/compute() notifications. The main difference between the two is the control flow of the sender - a call blocks and waits for a compute result, where as a send/process does not block. To simplify the internal architecture of the runtime system by reducing two communication paths to one, there is implemented call()/compute() with a series of coordinated send()/process() notifications, with additional system support to transform the first into the second. This transformation and implementation is entirely automatic and invisible to the application engineer. From the engineer's point of view, call and compute ports operate as regular method calls.

[0270] For every call port s or compute port t defined in the operator specification, the application code corresponding to that operator will have the corresponding two methods: call_s() is system implemented but available for use in the application logic, or compute_t() contains the application engineer's implementation of how to calculate results of a specific call and return the result. This method is invoked under cover of the system version of the compute() method.

[0271] Fig. 18 shows a process 1800 of calls that implement call/compute functionality via send/process functionality. An arrow indicates the direction of data flow. The task-instance that issues the call the Caller task-instance and the task- instance that computes the result of the call the Callee task-instance. Suppose the Caller task-instance call port is s and the Callee task-instance compute port is t. The protocol is as follows. The Caller application code invokes call_s(data) for call port s. The port and data are passed to the Caller call() system code. The Caller system code converts its arguments into a message and issues a normal asynchronous send() notification to the Callee and then blocks. The Callee process() system code receives the notification and calls the Callee compute() system code. The Callee compute() system code calls the application compute_t(data) on the Callee application code for compute port t. The application code computes the result of the input data and returns the result of the call to the Callee compute() system call. The Callee compute() system call then packages the result in a message and sends the result back to the Caller task-instance via a targeted send(). The message is received by the Caller process() method. The result is extracted from the message and passed to the blocked Caller system call(). The Caller system call() method unblocks and returns the result to the application Caller call(). Processing continues from that point on.

[0272] Depending on the runtime graph, the Callee system send() may arrive at more than one Callee task-instance. In this case, the first Callee result that arrives back at the Caller is

used and the rest of the results are discarded. In case the first answer is an error, the final response will be an error. The mechanism to simulate correct behavior for synchronous call()/compute() using the asynchronous send()/process() relies heavily on three things: the compiler rewrite of parts of the specification (operators, and dataflow), the code injected in the Auto_Op classes generated by the code generator for both Caller and Callee operators, and two special system classes Caller and Callee, from which the code generator will force inheritance.

[0273] The rewrites of the specification are as follows. Autocoder, automatically and invisibly to the engineer, injects new ports for each call/compute link in the dataflow (Caller, s) → (Callee, t): a new outgoing notification port s_request_start for the Caller operator; a new incoming notification port t_request_start for the Callee operator; a new outgoing notification port t_reply_end for the Callee operator; and a new incoming notification port s_reply_end for the Caller operator. The following links are added to the dataflow specification: a new link (Caller, s_request_start) → (Callee, t_request_start); and a new link (Callee, t_reply_end) → (Caller, s_reply_end).

[0274] The code generated by Autocoder's code generator will automatically inherit from two special classes Caller and Callee, that embed some of the basic mechanisms for the implementation, as follows: Caller Code: The application Caller code invokes call_s(data) with a given data argument. The method calls the generic system call() method of the Caller superclass 1900, shown in Fig. 19. The system method first creates a unique identifier for the call. The code then constructs a system event object payload that contains the necessary information for the compute to be accomplished - the call id, the application arguments of the call, and the caller's instance identifier for the return targeted send that will return the answer to the compute task-instance call. The system call code then creates a queue to process the result and places the queue in a dictionary. The code then invokes the normal send() notification to the compute task-instances through its special port newly created s_request_start. Finally, the system call blocks on the queue and waits for a response. Eventually, one of the compute task-instances which received the request will send back an answer. The answer will arrive as a normal process() notification on the newly created s_reply_end port. The logic attached to this automatically generated port will be automatically generated in the Auto_Op class. The logic will route the response to the general_process_reply_end() system method of the Caller class. This method unpacks the reply and puts the result onto the waiting queue. Placing the result in the queue wakes the blocked caller, which retrieves the result of the computation from the queue and returns to the application code. Processing continues as normal from this point.

[0275] Subsequent responses to this call will be simply ignored. Fig. 19 shows a sketch of the Caller class for the implementation of call/compute using send/process messaging.

[0276] Callee Code: The send notification on the port `s_request_start` is received by one or more task-instances on their incoming notification port `t_request_start`, according to the dataflow specification and the instance graph. `t_request_start` is a normal asynchronous input port, yet unknown to the application developer. The logic executed on that port is automatically generated in the `Auto_Op` associated with the Callee. The logic involves calling a generic method of the class Callee named `general_process_request_start()`. The system level code `general_process_request_start()` unpacks the contents of the process notification and then calls the specific application compute method. The compute method yields a result (or an error), which is packaged and returned to the caller via another send notification on the newly created `t_reply_end` port. Fig. 20 shows a sketch of the Callee class 2000.

[0277] The code generated for each specific `Auto_Op` for both Caller and Callee operators in order to make this mechanism functional is subsequently described.

[0278] Provenance in the Dataflow. A common computational pattern in dataflow systems utilizes the history - the provenance - of message data as it flows through the system. Provenance is useful for logging, debugging, and performance analysis. Provenance is also used by the logic of some application code (e.g., windowing, timeouts). The provenance information is structured as a historical sequence of triples (task, instance, timestamp) maintaining some (but not all) of the task-instances whose execution triggered that given message.

[0279] For low-code design, there is automatic support for provenance maintenance. In Autocoder, the tasks whose instances are required to perform stamping are marked with an additional property to the task description in the dataflow specification, as shown in Fig. 21. Fig. 21 shows a task 2100 that declares the provenance property.

[0280] If the provenance property is true, then for any message from any task-instance corresponding to that task, leaving from a send port of the task-instance, a stamp (task, instance, timestamp) is added to the list of provenance information associated with the input message. If the provenance property is false, the output message simply carries the provenance of the input message that triggered the current method.

[0281] Operator code can access the provenance through an API on the Event class. The API can return the last provenance record, the entire provenance (via an iterator), or return a filtered subset based on the most recent provenance records that belong to a particular task or operator.

[0282] Error Handling. Error handling can be a tricky part of operator implementation. In addition, error handling conflicts with low-code design principles because many lines of code may need to be written to handle various errors.

[0283] To help alleviate this problem, as part of the system implementation, all calls to the application code are wrapped with error handling code. Upon the generation of a runtime error, the system implementation will attempt to automatically execute an application defined `recover_from_user_error()` method for the operator, if the application developer has implemented this method. The `recover_from_user_error()` method receives both the message that was passed to the task-instance that generated the error and the exception object as arguments. The `recover_from_user_error()` application method's role is to revert the potentially inconsistent messages being sent already, and to revert the internal state of the task-instance to a consistent state. If the method does not exist, or is unsuccessful, the task-instance is blocked in what we call a zombie mode, where only system messages are allowed to be processed, but not application messages.

[0284] Optimized Single Process Application Execution

[0285] The default semantics are easy to understand by the domain expert and engineer. But the implementation of actors has a set of resource costs during execution. First, one thread per instance is allocated. Second, communication between threads requires message queuing and de-queuing. Third, locks for input messages are over-utilized. The immutability of messages eliminates problems of side-effects to messages bleeding into other operators, but it requires a message copy.

[0286] In some cases these resources are too much or too little for a given operator, and some of the expensive operations are not needed. Autocoder provides several better optimized variations of the default implementation described above, without requiring the engineer to write additional code. In some cases, optimizations can be done without any additional developer input. In other cases additional information is required; such input is given as annotations on the existing specifications. The additional information described now is only for performance improvement and is not needed for correct execution.

[0287] Minimizing Threads via Passive Execution Strategy. The default application semantics and, as a result, the default `TaskInstance` class implementation, requires all processing of the incoming messages of a task-instance to be executed on a separate internal thread. Some operator implementations make explicit use of this thread, and as result they are as marked as active. The logic and the code of the other operators, declared (explicitly or by default) as passive, is agnostic of which thread executes its code.

[0288] A task associated with a passive operator has the choice of being executed with as active (with internal thread) or passive (without internal thread). In the active execution strategy, two threads are involved in the sending of a message and the processing of the message, the sender task-instance thread executes the send() call and enqueues the message into the receiver's internal queue. The receiver task-instance's thread dequeues messages and executes the corresponding process() methods. In the passive strategy, the sender task-instance thread executes both the send() and the process(), and no queue is involved in between.

[0289] The PassiveTaskInstance, inheriting from GenericTaskInstance, is an alternative superclass for the application operators that contains the passive execution strategy implementation. While executed as either active or passive, the code of the application operator implementation does not change. The same externally observable semantics are implemented by the system. However, the internal execution strategy changes. To effect this change, the superclass of the operator implementation is automatically and invisibly changed by the Code Generator from the TaskInstance to the PassiveTaskInstance class.

[0290] The implementation of the passive execution strategy in the PassiveTaskInstance class is simple. During initialization the class does not create a thread for the task-instance. When messages arrive at the process() method, the messages are not queued, but processed immediately by the sender thread. Note that locks are still taken to process messages because two different external threads may simultaneously send messages to the receiving task-instance and thus simultaneously execute process().

[0291] Syntax of Execution Strategy Request. In order to minimize resources, Autocoder executes every task whose operator is passive as a PassiveTaskInstance object. However, the application developer has the freedom to request to overwrite this decision, requesting an active implementation for such a task. The request needs to be expressed using the following syntax in a specification called local optimization plan:

```
task:
  name: A
  execution:
    strategy: active
```

[0292] Note the asymmetry between operators and tasks. Operators declare to be either active or passive to describe their implementation. Tasks can request to be executed as active or passive in order to achieve some optimizations goals. Not all four combinations are legal. This logic is expressed in a compatibility matrix, shown in Table 7.

[0293] Table 7. The compatibility matrix for mixing active and passive execution strategies for operators and tasks.

Operator	Task	Result
Passive	Passive	Passive
Passive	Active	Active
Active	Active	Active
Active	Passive	(Illegal)

[0294] Note that executing messages on the incoming thread rather than the local actor thread changes the order of message processing to a depth-first order, rather than in parallel as it would be for normal actors. Since order of message processing is not guaranteed by the semantics, this change is acceptable.

[0295] In the running example, assume that the camera and crop tasks are co-located together in the same process, and the task crop is passive. The thread for the task- instance (camera, building/1/room/1/door/1/) captures an image from the device and sends it to (crop, building/1/room/1/door/1/). The image outgoing message is sent to the object PassiveTaskInstance corresponding to the task-instance (crop, building/1/room/1/door/1/). Because the crop task is passive, the same thread processes both the method of the camera task and the associated method of the crop task, for the same instance. The more complex case where camera and crop are located in different processes is discussed in the section on the distributed runtime.

[0296] Increasing Throughput. In the opposite case from reducing resources, one may discover a computational bottleneck in the task-instance graph. In the case that the solution to the bottleneck is more thread resources, the pool of threads processing messages for an active task can be increased from its standard 1 to a larger number. As messages arrive and are queued, available threads from the pool are assigned to process them. The thread-pool size is specified in the local optimization plan using the following syntax for a task named A with thread-pool size 8:

```
task:
  name: A
  execution: strategy: active
  threads: 8
```

[0297] The number of threads is decided statically on a per task basis. It is interesting to allow more flexibility, and allow finer granularity (e.g., allow various number of threads per task-instance), or to choose the number of threads dynamically.

[0298] In the absence of a threads declaration, an active task is executed with a single thread. Increasing the number of threads that execute a task-instance also changes the order of message execution, since threads compete to process messages. Because the order of message processing is not guaranteed by the semantics, this change is acceptable.

[0299] By default in case of multiple threads the message processing lock is set. This limits throughput. The cases where this problem can be overcome are discussed subsequently.

[0300] Minimizing Locking. Another potentially unnecessary cost is the lock taken by a task-instance for every processed message. In many cases the lock is not necessary. A common case, but not unique case, are operators that are purely functional and do not carry internal state from one message to the next.

[0301] In order to be able to optimize away the unnecessary locks, there is additional information for the semantics of the operator. The *has_internal_state* boolean property, declared in the semantics subsection of the operator specification, declares that the operator maintains internal state. The default value for the property is true.

[0302] In this sketch of the operator specification declaration, the other declarations have been replaced by an ellipsis.

```
operator:
  name: A
  ...
  semantics: has_internal_state: false
```

[0303] The lock taken by each message processing of a certain task-instance can be eliminated in the following cases: if the task is an active task with number of threads being 1 or if *has_internal_state* of the operator of the task is false. Some subtle cases make an operator stateful, and hence making this optimization impossible. An operator will be stateful in case the following kinds of data are kept as fields of the operator class, and are not declared thread local: data that is only needed to pass information to one method to another method, both collaborating towards processing a single message, but not passing any information across messages, or data of an instance state.

[0304] In general, any application data stored in an operator class as fields that is (a) non-constant and (b) and not declared as thread local automatically makes an operator stateful and

hence lock elimination is not possible⁵. As result, if performance is of concern, developers need to design and implement such details with care.

[0305] The locking decision is taken statically, per task basis, not on a task-instance basis, or even less dynamically on a per message basis. The *has_internal_state* property of the operator is passed by the dataflow compiler to the GenericTaskInstance object as a system parameter at initialization time. This parameter, in conjunction with the active vs. passive and thread parameters, drives the runtime locking behavior.

[0306] Autocoder Application Construction and Execution

[0307] In the centralized case, four tools of the eight described previously are particularly relevant. The design tool, dataflow compiler tool, code generator tool and single process execution tool are described in this section. The next section describes additional aspects of dataflow relevant to distributed computing, the build tool, the install tool and the distributed version of the execution tool.

[0308] Design Tool. Autocoder offers two methods for the domain specialist to specify the application, as shown in Fig. 22: a design tool that visualizes and allows editing some of the components of the specification (the entity ontologies and anchored dataflow). The second method is the direct specification via a set of text files that conform to the application specification. The operator algebra is not interesting to edit visually, and very likely the instance graph is the result of an external tool. Fig. 22 shows an image 2200 of a domain expert modeling the specification using a design tool. Fig. 23 includes an image 2300 that represents the domain expert modeling the specification using a design tool.

[0309] As part of the design of the specification, the domain experts and engineers choose external operators libraries, in our formalism. Each Autocoder library package of a self-consistent set of operators descriptions and the respective code. Example of such libraries can be implementations of various sets of sensors and actuators. In addition, a set of basic operators are provided that are likely needed in most IoT applications.

[0310] The design tool integrates the requested external operator libraries into the specification. The domain expert models the target anchored dataflow specification. When done, he/she publishes the specification to make it available for subsequent processing.

[0311] Dataflow Compiler. The dataflow compiler performs three primary functions. The first function is the validation of the specification published from the design tool, as shown in Fig. 24. The second function is the injection of system functionality to produce an extended dataflow, as shown in Fig. 25). No application code is modified by the dataflow compiler.

Finally, the compiler prepares the application defined customization for the runtime use, as shown in Fig. 26.

[0312] Fig. 24 shows a validate process 2400 that analyzes the specification and produces a document containing a list of problems, if any. Dataflow Validation. The dataflow validates the specification as correct through large collection of syntax and consistency rules, as shown in Table 8. Consistency rules are somewhat subtle. The rules must be strict enough to prevent a specification crashing dataflow compilation and execution, but the rules are loose enough to allow rapid development. For example, an operator declared in a task declaration must exist, otherwise compilation will fail. However, the operator ontology may declare operators that are not used by any task. These extraneous operators are ignored by the dataflow compiler. During validation the default values are added for the various optional fields in the specification.

[0313] Table 8. A sample of consistency rules for the verification step.

Specification	Rule
operator	Implementation section declares language
dataflow	Task operator exists
dataflow	Task scope exists in entity ontology
dataflow	Task and send port used in connections exist
dataflow	Task and process port used in connections exist

[0314] Fig. 25 shows an injection process 2500 that takes the anchored data flow and operator ontology and injects additional specifications to generate the extended data flow and extended operator ontology.

[0315] Operator and Dataflow Rewrite. The dataflow compiler also injects additional functionality. There are examples of adding functionality described including adding local process controllers to provide process-level task instance control and transforming synchronous calls into a series of asynchronous messages. Each such step takes a version of the specification as input and produces an enhanced version as output.

[0316] Adding Local Process Controllers

[0317] A single controller (called process controller) starts, pauses, and stops the set of tasks running in a given process. In this step this controller is made explicit.

[0318] The controller is a predefined system task in the system operator library. Among the functions of the process controller are (a) control the four states of execution for each task instance, (b) process notifications of any runtime errors by any of the task instances in its process, and (c) investigate the status of a task instance during execution.

[0319] In a complex distributed environment the process controller does not act in isolated manner, but under the control and guidance of a global controller, which in itself probably runs

under the surveillance of a human operator. In that case, each one of the actions described here performed by the process controller task is a consequence of processing a system message from such a global controller.

[0320] A sketch of the ProcessController operator specification in the system library is as follows:

```
-operator:
  name: ProcessControllerOperator
  input-ports:
  -process:
    name: error
  output-ports:
  -send:
    name: start
  -send:
    name: pause
  -send:
    name: restart
  -send:
    name: stop
  -call:
    name: status
  implementation:
  language: python
  strategy: active
  module: autocoder.library.operator.process__controller
```

[0321] The dataflow rewrites does the following changes: adds a new task to the dataflow. A sketch of the task is as follows:

```
- task:
  name: process__controller
  operator: ProcessControllerOperator
```

[0322] For every task T in the dataflow add the following system ports:
output-ports:

```

-send:
    name: error
input-ports:
-process:
    name: start execution: fast
-process:
    name: pause execution: fast
-process:
    name: restart execution: fast
-process:
    name: stop
-compute:
    name: status

```

[0323] For every task T add the appropriate links between pair is same-name ports $p \in \{\text{start, pause, restart, stop, status}\}$ ($\text{local_controller}, p$) $\rightarrow$ (T, p). For every task T add the link (T, error) $\rightarrow$ ($\text{local_controller}, \text{error}$).

[0324] All the above system calls are executed by Autocoder's runtime as normal messages. However, in order to have the expected behavior, some of such system ports need to be fast ports, previously described.

[0325] Transforming Synchronous calls into Asynchronous messages

[0326] As previously described, call/compute is implemented using send/process. The dataflow compiler injects the ports and connections required to support the system code. The rewrites of the specification are repeated here for readability. Autocoder, automatically and invisibly to the engineer, injects new ports for each call/compute link in the dataflow (Caller, s) $\rightarrow$ (Callee, t): a new outgoing notification port $s_request_start$ for the Caller operator; a new incoming notification port $t_request_start$ for the Callee operator; a new outgoing notification port t_reply_end for the Callee operator; or a new incoming notification port s_reply_end for the Caller operator.

[0327] In addition, the following links are added to the dataflow specification: a new link ($\text{Caller}, s_request_start$) $\rightarrow$ ($\text{Callee}, t_request_start$) and a new link ($\text{Callee}, t_reply_end$) $\rightarrow$ ($\text{Caller}, s_reply_end$). At the end of the rewrite, the link between (Caller, s) $\rightarrow$ (Callee, t) is deleted.

[0328] Customization Tool via Application Parameters. The last step in the dataflow compilation is dedicated to processing application input with respect to customization of the

executable code, shown in Figure 26. The parameterization process 2600 takes the takes a variety of data and generates task-instance specific parameters. Generic operators from existing libraries are customized in applications in different ways, each with different constraints and assumptions.

[0329] For example, consider the instantiation of the CameraPiSensor operator. Depending on the pairing of this operator with a particular instance, the parameters may be different. Each camera can have different initialization for resolution, adjustments to coloring, etc. Moreover, other parameters are determined at other levels of abstraction. For example, a parameter maybe set for all instances of an operator, regardless of the particular instance. The application parameter declaration can occur at multiple levels of abstraction: at the operator level, at the task level, or specified at a task instance level, in order of increasing precedence. Each parameter declaration is simple a pair (name, value) pair, with arbitrary complex values.

[0330] For the running example, a parameter at the operator level is declared in the operator parameters file with the following syntax:

```
- operator-parameters:
    operator-name: CameraPiSensor
    parameters
    -parameter:
        name: width
        value: 640
    -parameter:
        name: height
        value: 480
```

[0331] The same parameter at the task level is declared in the task parameters file, with the following syntax.

```
- task-parameters:
    task-name: camera
    parameters
    -parameter
        name: width
        value: 640
    -parameter:
        name: height
        value: 480
```

[0332] The same parameter at the task-instance level is declared in the task-instance parameters file, with the following syntax.

```
-task-instance-parameters:
  task_name: camera
  instance_id: build/1/room/1/door/1/
  parameters
    -parameter
      name: width
      value: 640
    -parameter:
      name: height
      value: 480
```

[0333] Parameters overwrite themselves from the more general to the more specific, so task-instance has the higher priority, followed by task, followed by operator with the lowest priority.

[0334] The dataflow compiler gathers the three parameter files and generates a new task-instance application parameters internal file that lists for each task-instance in the runtime graph the final determined value of each parameter. Upon initialization of the task-instance objects, the parameters are automatically set as local variables in the task-instance state. The need to specify parameters at the task-instance level is the only place in the dataflow compiler that requires knowledge of the instance graph. This case is relatively rare and typically happens in debugging and simulation (e.g. make a particular instance sensor behave erroneously), but not in production systems, which appear to have homogeneous parameters settings.

[0335] Finally, some parameters values are file names that refer to files in an application specification directory. Such files references are moved into the right location by the build tool, so that the data in the files is available at runtime to the associated task-instances.

[0336] Code Generator Tool. In the previous section the Autocoder's predefined runtime code was described, with all the bells and whistles needed to implement all possible semantic choices, and several possible optimizations.

[0337] The next tool, the code generator helps bridge Autocoder's runtime code with the application code. Code is generated at three different levels.

[0338] The auto stubs contains code expressing a trivial implementation that does nothing. This code is replaced by application or library developers with the intended implementation.

[0339] The auto bridge code bridges the application operators code above and the runtime classes that support their semantics. Both the library bridge code and the library stubs take the

operators file as input, before any of the rewriting described above has been done. This file includes the extensions describing the semantics of the operator (e.g. active or not) that are needed for the code generation. In addition, the bridge code takes also the local optimization file as input.

[0340] The system parameters generation outputs the system parameters required for the generated code above to work correctly and optimally.

[0341] Autocoder generates Python code, the same programming language as used by Autocoder's runtime. In case a new runtime language is added, the code generator tool needs to be extended to generate the new language as well. In addition a new runtime must be added.

[0342] Auto Stubs. For every operator op Autocoder generates skeleton code, a class Op that includes several parts: an automatic inheritance declaration to the bridge class Auto_OP; required library imports from the pip field; for each process port p: a new empty method process_p() (to be implemented by the application developer). In case one of those methods is missing the class Op will fail to load; for each compute port c: a new empty method compute_p() (to be implemented by the application developer). In case one of those methods is missing the class Op will fail to load; optionally, for clarity, the class contains, in comments, examples of the available methods send_s() corresponding to each send port s and call_c() corresponding to each call port c; optionally, for additional help, the stub for the optional method recover_from_error() is also generated in comments. For example, given the operator specification of the *CameraSensor* operator shown in Figure 11, the dataflow compiler generates the skeleton code bellow:

```
# "picamera==1.13" # "Pillow==8.3.2"
from autocoder.infrastructure.core.event import Event
from autocoder.library.schema.image import Image
from .auto_tasks import AutoCameraSensor
"""
    CameraSensor
    process_heartbeat(self, Event e)
    self.write_image(Event(...))
    """
    def init (self, *params, **kw_params):
        super(). init (*params, **kw_params)
    def process_heartbeat(self, Event e):
```

pass

[0343] The code of the classes Op is replaced by the library developer, using their favorite programming environment.

[0344] Auto Bridge Code. For every operator op Autocoder generates bridge code between application code and the system runtime, a class Auto_Op that includes several parts. These include an inheritance declaration from either PassiveTaskInstance or ActiveTaskInstance, depending on the value of the implementation strategy field of the dataflow task T using the operator Op. These include an inheritance declaration of Auto_Op from the runtime class Caller (in case the operator has at least one call port) and/or from the runtime class Callee (in case the operator has at least one compute port), in order to guarantee correct execution of calls/compute synchronous messages. These include, for every process port p a new abstract method process_p() is generated in order to force application developers to specify the application logic of that port. These include, for every compute port c a new abstract method compute_c() is generated in order to force application developers to specify the application logic of that port.

[0345] For every send port p a new method send_p() is generated, with the following code:

```
send_p(...):
    send("p", ...)
```

[0346] The port-specific method send_p() is routed the generic version of the method send(). For every compute port c a new method call_p() is generated, with the following code:

```
call_p(...):
    call("p", ...)
```

[0347] The port-specific method call_p() is routed the generic version of the method call(). For every call port c a new method with the code:

```
process_c_reply_end(...):
    generic_process_reply_end("c", ...)
```

[0348] The port-specific processing of a call response is routed the generic version of the method. For every compute port c a new method with the code:

```
process_c_request_start(...):
    generic_process_request_start("c", ...)
```

[0349] The port-specific processing of a compute start is routed the generic version of the method.

[0350] Generation of System Parameters. This part of the code generation assembles all the system defined task properties and converts them into system parameters that will control the

correct and optimal execution of the runtime graph. The final values are stored in system parameters file.

[0351] For every task T in the extended dataflow specification, the following parameters are registered for it, with their respective values: *has_internal_state* of type boolean; *reads_immutable_instance_state* of type boolean; *reads_mutable_instance_state* of type boolean; *writes_mutable_instance_state* of type boolean; *threads* of type integer to describe the number of threads, in case where the execution strategy of a task is active; and *provenance* of type boolean.

[0352] The first four parameters are obtained straight from the operator semantic description of the operator Op corresponding to the task T. The correct default values for those parameters (if they were not explicitly declared) have been given during the validation stage of the dataflow compiler. The last parameter is obtained from the local optimization specification of the task T. Those parameters will be given to each instance of a runtime graph node as part of the object initialization.

[0353] Execution tool

[0354] Initialization of the runtime graph. The first step in the execution is the instantiation of the runtime graph, following closely the formal definition. The input specifications are: the extended dataflow, (b) the extended operators, (c) the system parameters, (d) the application parameters, (both generated by the compiler) and finally (e) the instances file. The result of Algorithm 1 is a set of connected instances of *GenericTaskInstance* that is ready to be operational. This set includes an object that is the process controller object. Starting the controller object will trigger the start of the entire runtime graph. The execution is now in progress, task-instances will start producing and sending messages to each other.

[0355] Distributed Execution of Applications

[0356] Previous sections gave a formal definition for an application specification based on the notion of a runtime graph, and described a centralized execution for an application. In that case, the entire application, i.e. all task-instances in the runtime graph, were executed within a single process. While the centralized case is effective for debugging a small application, it does not scale to larger apps. Another alternative would be to execute every single task instance in its own process. Besides the large number of processes, this alternative suffers from performance issues because message serialization/deserialization occurs when messages cross process boundaries, i.e., for every message sent. In reality, applications split into parts, each to be executed in a separate process, and communicating among them and using various communication protocols (e.g. MQTT, HTTP). The execution is typically distributed on a

variety heterogeneous hardware, ranging from the cloud, all the way to small microprocessors connected to analog devices.

Algorithm 1 Instantiation of Runtime Graph

Require: maintain H , a hash table of all task-instance objects, indexed first by task, then by instance

```

for each task  $T$  do
    retrieve the system parameters  $S$  of the task  $T$ 
    for instance  $I$  in  $\text{scope}(T)$  do
        retrieve the application  $A$  parameters corresponding to the pair  $(T, I)$ 
        initialize a new instance of the class implementing the operator  $O$  of the task  $T$ 
        with  $A$  and  $S$ 
        add the new object to the hashtable  $H$ 
    end for
    for each edge  $(T_1, p_1) \rightarrow (T_2, p_2)$  labeled  $R$  do
        for instance  $I_1$  in  $\text{scope}(T_1)$  do
            for for each instance  $I_2$  in  $\text{scope}(T_2)$  such that  $I_1 \rightarrow I_2$  with label  $R$  do
                add a new connection from the task instance  $(T_1, I_1)$  on port  $P_1$  to the task
                instance  $(T_2, I_2)$  on port  $P_2$ 
            end for
        end for
    end for
end for

```

[0357] In order to understand a distributed execution of a runtime graph, the answers to the following questions are obtained: Q1 How is the runtime graph split into partitions, each partition being executed in a separate process? Q2 What are the available processing units for executing the application? Q3 On which CPU does each partition/process execute? Q4 How do various processes communicate? The answers to the above four questions form the distributed application execution plan. In this section there is described an answer to the above questions in the form of a formal definition of a distributed execution plan. The distributed execution is then described.

[0358] Two other important questions with large implications on the cost metrics and build/install mechanisms are answered: Q5 What is the hardware and software of each processing unit (CPU)? Q6 What are the network protocols used to communicate between each pair of communicating processing units? These latter two questions are answered separately from the first four. While the answer to the first four questions form the logical distributed execution plan of a runtime graph, the answer to the additional two questions deals with the low level decisions about the implementation of the heterogeneous distributed execution, and form the physical execution plan.

[0359] There are three points to observe about our notion of distributed execution plan. First, no matter of the choice of the distributed execution plan, the resulting application should have

the same semantics. The distribution should be seamless and invisible to any user of the application. Different execution plans impact only the metrics associated with an application: the cost, reliability, availability, performance, etc., but all execution plans have the same semantics.

[0360] Second, in the formalism the distributed execution plan, logical or physical, is described only in terms of the application specification, and is completely independent of the set of instances the application operates on. Such a instance-dependent execution plan brings a huge complexity, and it is unclear if such flexible/dynamic execution plan is beneficial in any way.

[0361] Third, in the current formalism, the answer to the questions above is static and predetermined before an application execution is in flight. An "optimal" plan is calculated, manually or automatically, that makes the best compromise between the various dimensions: financial cost, response time, reliability, availability, security, etc., given the constraints and priorities of a particular application.

[0362] While the first observation needs to remain true in any future work, the other two do not. Possible extensions of this work may relax those two limitations. It is possible to conceive that parts of the runtime graph migrate dynamically, and on an instance basis, not homogeneously, based on the current field conditions of hardware, software, data and computation.

[0363] In general, the answers to the design questions of distributed computation are guided, and constrained, by the requirements of a particular class of applications. These requirements cross many dimensions. For example, many sensors must operate near the source of a signal, thus requiring geographic distribution. Applications require a degree of reliability and availability, and a minimum performance in response time and throughput. The design considers constraints on power, connectivity and cost. And designs have requirements for security, privacy, mobile access, and regulatory requirements. The design difficulty comes from the interactions between the solutions to these requirements.

[0364] This section applies to IoT applications, which is subsequently described formally. Abstractly, the runtime graph can be split in several ways. One possibility is to partition the instances (and their associated data) but not the tasks. Effectively an entire copy of the system is replicated for each partition of the instances. Another way splits the computation but not the instances. In this arrangement, for a given computation, data is sent to the location where a part of the computation can be performed. An "optimal" plan is almost always some combination of these two extreme potential solutions.

[0365] Another question concerns the set of processing units available for execution. One class of architectures of IoT application is cloud-based: sensors and actuators are executed where sensing/action was needed, and all other computation is done in the cloud; those are the only two types of processing units available to an application. This arrangement offers several benefits, such as taking advantage of the dynamic assignment of resources to computation in the cloud (auto-scale out), lower human costs and higher flexibility for product use and sharing through the web, increased reliability and lower operations cost though the sharing of operations personnel across many applications.

[0366] Another class of architectures is termed “edge” computing. Originally edge computing focused on limitations by placing hardware and software in network communications equipment. Now edge computing refers to any placement of hardware and computation in between the lowest level hardware of sensors and actuators and the highest cloud layer.

[0367] Obviously, adding additional layers of computation and distributing the processing across them significantly increases the complexity of the architecture, especially if computation is done by a heterogeneous network of hardware/software stacks, communicating through heterogeneous network protocols. However, the rationale for this distribution is powerful and beats the additional complexity: First: feasibility (e.g. a sensor can only be executed on a small chip, data analytics can only be executed in the cloud). Second: cloud execution imposes the harsh constraint that the sensors and actuators need to be in constant network communication with the cloud, which is clearly not feasible in many environments, especially mobile environments. Even if such communication is theoretically possible, the HTTP protocol is very power hungry, and that imposes hard constraints on the available power, which again are unfeasible in many cases (e.g. sensors in an agricultural field with no power lines). Of course there are additional reasons (e.g. cost, security, performance, availability) make edge distribution a non-disputable architectural choice.

[0368] In edge architectures, computation is executed on a set of available CPUs that are heterogeneous from all points of view: geographical location, capabilities and properties, security requirements, and ownership. This definition covers a whole spectrum of computational devices from small chips all the way to the cloud.

[0369] In order to answer the question of what are the available CPUs for executing an application, it helps to simplify by splitting the answer in two levels of abstraction. First, we need to define where processing occurs (a logical CPUs) and then, separately, to define which particular hardware/software (a platform) supports each particular logical CPU. By a logical CPU we understand the following.

[0370] A logical CPU is a self-contained computing device, with a unique access point, that contains part of the hardware and runs part of the software needed for a full application. Sensors and actuators can both be abstracted as being executed on logical CPUs. Even if analogue in nature, eventually the analog signal is turned into digital information on a CPU. Similarly, note that a single "cloud" entry point forms in our world a logical CPU, despite of the obvious hardware distribution that happens inside the cloud. In fact, our notion of logical CPU provides an abstraction over a wide range of devices, from a small-scale microchip like an ESP-32, to devices such as an Arduino/Raspberry-PI (or laptop, smart phone, or server), to a cloud server (which may represent a set of auto-scaling machines).

[0371] The question of available CPUs can be rephrased at the logical level as more detailed questions: What is the set of logical CPUs available for the distributed execution of an application? Which pairs of logical CPUs are able to communicate, and what is the topology of the graph they form in this manner?

[0372] What is the relationship between the instances graph and the logical CPU graph? At the physical level, there is a particular choice for each logical CPU in terms of particular hardware/software that is supporting the logical CPU, and a particular network protocol that implements communication of each communication edge between logical CPUs. Note that the distribution and communication parts of the execution plan focus on the logical CPUs, not on the physical platforms.

[0373] The computation is placed on the logical CPUs. Given a graph of available logical CPUs (processing nodes linked by data communication edges), different execution plans place different parts of the application at different nodes in the logical CPU graph. One popular architecture places as much computation as possible in the cloud. At the opposite spectrum is the heuristic of placing all computation as "close" (either geographically, or in number of software "hops") as possible to the source or target of the data. Typically, neither extreme works well. In practice, a careful distribution of the partitions of the runtime graph into the CPU nodes need to be chosen, crafted to satisfy the application requirements, and to best satisfy the application priorities by a compromise between various cost metrics.

[0374] To determine how partitions communicate, given a partitioning of the runtime graph into sub-graphs that execute in different processes, each being potentially located on a different CPU, the data processing system determines the pattern/protocol by which non co-located partitions send messages to each other. Logically, possible answers are point-to-point protocols (e.g., HTTP), or publish-subscribe protocols (e.g. MQTT, Kafka). The choice of the protocol is relatively independent on the past two decisions. A given partitioning and a given distribution

of partitions can be implemented and supported by various communication mechanisms and protocols.

[0375] For many applications, including Internet of Things and large-scale distributed applications, publish/subscribe is used as the main backend communication platform.

[0376] Publish/subscribe is a popular communication system in IoT because offers a simple communication mechanism for systems with a dynamic number of participants, since the sender and receiver of messages do not have information about each other (that is, communication is anonymous). MQTT is a popular publish/subscribe system with many open source implementations that are available across a wide range of devices. Publish/subscribe systems are generally organized around a client/server architecture. Different implementations of publish/subscribe offer a wide range of guarantees and performance trade-offs, depending on the targeted application area. Another common backend platform architecture utilizes a collection of services on top of HTTP REST or other remote procedure call technologies. In general the specification of an application allows for any kind of communication between clusters.

[0377] Restrictive use case for IoT

[0378] The formalism and implementation described so far use a general purpose ontology with no restrictions on the set of entities and/or relations among them. In this section we restrict our solution to a particular ontology, common in the case of IoT applications, where most instances are related to physical objects in the real world, and the major entity relationship is a "has-a" relationship.

[0379] For this case, the following assumptions are made on the entities graph and on the instances graph: there exists a single relationship (called "has-a" in our example) in the ontology; the set of entities are connected through this unique relationship; the edges are directed from parent to child; the relationship "has-a" creates a tree on set of entities of the ontology; and the instance graph is also a tree based on the relationship "has-a", i.e. it has a unique root *rootinstance* and *rootinstance* $\in$ *instances* (*rootentities* where *rootentities* is the root of the entities tree).

[0380] Let us introduce additional definitions for a general tree. Those notions are used for both the entities graph, as well on the instances graph (both being trees). Given a node *n* let *ascendants* (*n*) be the set of nodes *m* such that it exists a (possible empty) directed path from *m* to *n*. Note that *n* $\in$ *ascendants* (*n*). Given a node *n* let *descendants* (*n*) be the set of nodes *m* such that it exists a directed path from *n* to *m*. Note that *n* $\in$ *descendants* (*n*). Given *k*

nodes $n_1, \dots, n_k$, the *lowest_ancestor* ($n_1, \dots, n_k$) is the node m that is an ancestor of all nodes $n_1, \dots, n_k$ and it has the longest path from the root to m . In this section, *ascendents* (), *descendents* (..), or *lowest_ancestor* (..), are referred to in reference to the "has-a" relationship, both on the entities tree, as well as the instance tree.

[0381] In addition to the "has-a" special relationship, more relationships are defined on the set of the entities and instances in the ontology, that will be used in the application specification: "part-of" is the inverse relationship of the relationship "has-a"; "contains" is the transitive closure of the relationship "has-a"; "contained-in" is the transitive closure of the relationship "part-of"; "id" is the identity relationship; "connected" is the union of "contains", "contained-in" and "id".

[0382] The only limitation in this section on distributed execution of dataflow application is that the dataflow anchoring use only the "connected" relationship, as defined above. In practice, this requirement means that a given message send by a task instance (T_1, i_1) will be received by another task instance (T_2, i_2) if and only if tasks T_1 and T_2 are connected in the dataflow graph and instances i_1 and i_2 are linked in the instance graph via the "connected" relationship (either "contains", or "contained_in" or "identical").

[0383] This limitation is not too restrictive for the case of IoT applications. In fact, it is highly improbable that a device will want to send information/messages/events to another instance that it is not connected to. At the extreme, if the communication between two unconnected instances is necessary, it will very likely communicate through a commonly connected instance (i.e. a common ancestor). In conclusion, we do not consider this a real limitation, but rather an accurate description of message communication in the IoT real world. However, all the mathematical definitions above will make the task of formally answering the four questions in this section much easier in the restrictive case than the general ontology case.

[0384] Generally the runtime graph is too large to fit in a single process, so the graph is split into smaller pieces, each being executed by a separate process. The solution relies on partitioning the set of tasks in the dataflow into a set of task clusters, each cluster having a unique name, as shown in syntax 2700 of Fig. 27.

[0385] Given an application specification that includes a dataflow with the set of tasks $\{T_1, \dots, T_n\}$, a clustering specification is a partition $C_1, \dots, C_n$ of the set of tasks $\{T_1, \dots, T_n\}$ into disjoint subsets, each cluster having a unique name. The syntax for declaring a task clustering is described by example in Figure 27 and illustrated in Figure 28.

[0386] Fig. 28 shows a cluster organization 2800 of tasks in the running example. Given a particular instance graph I , a clustering will naturally impose a partitioning of any runtime graph of the application that corresponds to the instance graph I . In the remaining of this section we will refer to a partition of the set of tasks as a cluster and a partition of the set of task instances in the runtime graph as a cluster instance.

[0387] To compute the cluster instances from a set of clusters, an additional definition is determined. Given a cluster $C = \{T_1, \dots, T_k\}$ that groups a set tasks, the *cluster_scope* (C) is the entity e such that $e = \text{lowest_ancestor}(\text{scope}(T_1), \dots, \text{scope}(T_k))$.

[0388] Table 9 show the *cluster_scope* for the running example.

cluster C	$e = \text{cluster_scope}(C)$	$\text{instances}(e)$
cluster_one	door	{building/1/room/1/door/1/, building/1/room/2/door/1/, building/1/room/2/door/2/ }
cluster_two	building	{building/1/ }

[0389] The *cluster_scope* identifies the common entity parent for a set tasks. At runtime, given an application specification, a clustering specification and an instance graph I , a clustering of the tasks in the dataflow graph imposes a natural partition on the task instances of the runtime graph, as follows. Given a cluster $C = \{T_1, \dots, T_k\}$ of the dataflow graph and an instance $i \in \text{instances}(\text{cluster_scope}(C))$, then the *cluster_instance* (C, i) = the set of *task_instance* (T, i') where

[0390] the task $T \in C$ the instance $i' \in \text{instances}(\text{scope}(T)) \cap \text{descendants}(i)$.

[0391] For the running example, Table 10 contains an enumeration of the cluster task-instance definition. Correctness. The *cluster_instance* function has two properties: disjointness and coverage. These two properties combined implies that *cluster_instance* forms a partition of set of task instances of the runtime graph. For Runtime-Partition Disjointness, if two cluster instances (C_1, i_1) and (C_2, i_2) have a non-empty intersection, then $C_1 = C_2$ and $i_1 = i_2$.

[0392] Runtime-Partition Coverage. Given a task clustering with p clusters, for the runtime graph RG corresponding to the instance graph I , *coverage* (RG, p, I) is defined as $C=1..p, i \in \text{instances}(\text{cluster_scope}(C)) \cup \text{cluster_instance}(C, i)$.

[0393] Table 10 shows enumeration of task-instances by cluster for the running example.

Cluster C	Instance i	$cluster_instance(C,i)$
cluster_one	building/1/room/1/door/1/	{ (heartbeat, building/1/room/1/door/1/), (camera, building/1/room/1/door/1/) }
cluster_one	building/1/room/2/door/1/	{ (heartbeat, building/1/room/2/door/1/), (camera, building/1/room/2/door/1/) }
cluster_one	building/1/room/2/door/2/	{ (camera, building/1/room/2/door/2/), (heartbeat, building/1/room/2/door/2/) }
cluster_two	building/1/	{ (crop, building/1/room/1/door/1/), (crop, building/1/room/2/door/1/), (crop, building/1/room/2/door/2/), (recognize, building/1/room/1/), (recognize, building/1/room/2/), (enrich, building/1/room/1/), (enrich, building/1/room/2/), (monitor, building/1/) }

[0394] (Runtime-Partition Coverage Completeness). Given a task clustering with p clusters, for any instance graph I $coverage(RG, p, I) = RG$.

[0395] The disjointness and coverage guarantees of *cluster_instance* insures that a clustering of the tasks in the dataflow graph imposes a unique partitioning of the runtime graph for any instance I . Each cluster instance is uniquely identified by a pair (C, i) where C is the task cluster and i is an instance such that $i \in cluster_scope(C)$.

[0396] Each cluster instance is executed in a separate process and, from the code build/organization point of view, the process has its own container. Task-instances that belong to the same cluster instance execute in the same process and communicate in memory as described in the previous section. Task-instances exchanging messages across different cluster instances communicate through inter-process protocols.

[0397] Available logical CPU and their communication topology are now described. The data processing system determines the set of logical CPUS available for the distributed execution of an application, the pairs of logical CPU that are able to communicate, and the topology of the graph they form in this manner. The relationship between the instances graph and the logical CPU graph is based on the following.

[0398] (Instances-CPU binding) Each logical CPUs is attached to an instance in the application's instances graph. This instance, with an associate processing power, is an active instance.

[0399] (Processing homogeneity) Given an entity E , either all instances of E have an associated logical CPU, or none.

[0400] ("Connected" communication) Two logical CPUs do not communicate unless their respective instances are "connected" in the instance graph.

[0401] Informally, each processing unit is always associated with a real world instance (e.g. a car, a room, a door, a building). No processing unit exists without such association. On the other hand, not all real world instances have an associated CPU, some do, some do not. Real world instances of the application can be either physical objects (e.g., a door) or virtual objects (e.g., an account). It is natural to assume that the CPU attached to a physical object is geographically located in the proximity of the object and shares much of the properties of the object (e.g. power availability, connectivity). For example, if sensors and actuators corresponding to a room are executed on a CPU of that room, then it is very likely that that CPU is physically located in that particular room. Non-physical instances (e.g., bank accounts) typically do not have an associated CPU, and hence are not "active", or, if they do, it is very likely that the associated CPU is a cloud node. This restriction is not imposed.

[0402] An all-or-nothing policy is used. Either all instances of an entity are active, or none are active. For example, in a building, either all rooms are active, or none. This homogeneous nature of instances of the same entity simplifies the task of scaling the application to a large number of entities and instances.

[0403] Two CPUs, attached to two uncorrelated and unconnected active instances in the real world, are not required to communicate data/messages to each other. CPUs attached to active instances communicate only if a relationship "connected" exists between the two instances (necessary, but not sufficient condition). Even if a two active tasks e_1 and e_2 are connected in the ontology E , this does not automatically imply that instances of those entities actually have a network communication and send data between their respective CPUs.

[0404] The solution based on the principles of imposing a binding all CPUs to instances of the application, and that of homogeneity within an entity, might appear over-restrictive. However, it has an advantage in terms of applicability, clarity and simplicity. In addition, the constraints of the processing model are not a problem but in fact a major feature because they naturally map to an efficient low-code design. Far from being too restrictive, every time such a constraint was violated, a bug was exhibited in the application design, and a faulty design of the entities and their relationships was highlighted. If an "unbound" CPU is required by an application (with no associated instance to "host" it), or if a communication between two non-connected real world entities is required, then somewhere we found a design problem of the entities, instances and their relationships. Typically an analysis highlights information that is necessary for the application, yet not captured by the current design.

[0405] Formally, the set of available logical CPUs and their connectivity naturally derives from the CPU graph defined as follows. Given an application specification that includes the ontology E , the CPU specification is a graph called CPU graph that is a sub-graph of E such that the set of nodes of the CPU graph is a subset of the set of entities E . We call those entities active entities. The set of edges of the CPU graph is a subset of edges of the ontology E via the relationship "connected". Those edges are network edges.

[0406] Fig. 29 shows a CPU specification graph 2900 including of the active entity graph and the connected relation in the running example.

[0407] Syntax. The syntax for declaring a CPU graph is the following:

```
-active-entity: entity:
    building
-active-entity:
    entity: door
-network-communication:
    entity1: building
    entity2: door
```

[0408] The CPU specification has follows these rules: (a) active entities are a subset of all application entities, (b) a network edge connects two active entities and (c) a network edge links two entities that are also linked in the application entities tree via the "connected" relationship.

[0409] At runtime, given an application specification, a CPU specification and an instance graph I , for each active entity e and for each instance $i \in \text{instances}(e)$ there is a logical CPU attached to that instance, labeled with $CPU(i)$. Additionally, for every pair of instances i_1, i_2 such that (a) there is a link $i_1 \rightarrow i_2$ in the instance graph (edges through the "connected" relationship); (b) both $\text{entity}(i_1)$ and $\text{entity}(i_2)$ are active and; (c) the entities $\text{entity}(i_1)$ and $\text{entity}(i_2)$ are connected in the CPU graph, then the network communication between $CPU(i_1)$ and $CPU(i_2)$ is possible.

[0410] In other words, the semantics of the definitions impose a the required network communication connectivity between instances. Fig. 30 shows a network communication connectivity 3000 in the running example. Each rectangle represents the CPU associated with the instance identifier.

[0411] **Placement of Partitions**

[0412] Given a set of cluster instances that form a partition of the runtime graph, and the graph of active instances, the next question is the placement of each cluster instance for execution within the available CPUs associated with the active instances.

[0413] The solution to the placement problem relies on the pairing between clusters and active entities. Given an application specification that includes the ontology E , a clustering specification, and a CPU specification, then a placement specification is a mapping $\text{placement}()$ between each cluster $c \in C \rightarrow e \in E$ where the following constraints are satisfied: e is an active entity, and $e \in \text{ascendants}(\text{cluster_scope}(c))$.

[0414] The syntax we use for describing a placement decision is exemplified in Figure 31. Fig. 31 shows a cluster placement decision 3100 for the running example. In general there are multiple possible placements. In the running example, only one placement is legal. The cluster *cluster_one* must be at the *door* entity because the operator of the camera task has a *false* value for the mobility property. The cluster *cluster_two* must be placed at the building level because the *monitor* task has scope *building*.

[0415] At runtime, given an instance graph I , each *cluster_instance* (C_i) is executed at the CPU (i') where $i' = \text{ascendants}(i) \cap \text{instances}(\text{cluster_placement}(C))$. Fig. 32 shows a placement 3200 of clusters in the CPU graph for the running example.

[0416] Correctness- for every cluster instance (C, i), the instance i' defined as above exists and it is unique. This theorem guarantees that the each cluster instance is correctly and uniquely assigned to a logical CPU.

[0417] Not all computations have the freedom to be executed on any CPU. In many cases there are strict bindings of certain operators and the CPUs that need to execute them. For example, the sensors need to be executed on a CPU associated to the instance the sensor is assessing, and not somewhere else. In many cases an operator implementing a complex data analytics operation can only be executed in the cloud, and does not enjoy the freedom to be placed and executed somewhere else. Such constraints are expressed via the *mobile* property of operators. This true/false value of the mobile property of an operator O indicates whether a task instance (T, i) whose task T is associated with the operator O can (or cannot) be executed on any other CPU other than the one of the instance i . In order to describe the value of this property, we extend the description of operators with an additional mobile field in the semantics section of the operator, with Boolean values. The default value for the mobile property of operator is *true*.

operator:

```

name: CameraSensor
...
semantics:
  has_internal_state: ...
  mobile: false

```

[0418] A cluster C containing only tasks whose operators are mobile can thus be placed at any active entity that is legal according to the definition. However, a cluster C containing at least one task T whose operator O is non-mobile cannot be placed at any other entity other than the entity that is the anchor of the task T in the application specification.

[0419] Cluster Communication

[0420] The Autocoder uses a publish/subscribe system to communicate between clusters. In this paper, we refer to the server as a broker, the term used by the MQTT literature. The Autocoder system makes the following assumptions about the communication between cluster instances running in separate processes: (Cluster-instance-pair broker uniqueness) given an application specification, a clustering specification, and an instance graph I , then for any pair of communicating clusters instances $(C1, i1)$ and $(C2, i2)$, there exists a unique broker such that any pair of communicating task instances $(T1, i3) \rightarrow (T2, i4)$ such that $(T1, i3) \in (C1, i1)$ and $(T2, i4) \in (C2, i2)$, the pair of task instances use this broker.

[0421] (Broker-CPU binding) brokers run on the logical CPUs in the existing application infrastructure (see above).

[0422] (Broker homogeneity) If a CPU attached to an instance i hosts an MQTT broker, then all the CPU attached to instances in *instances* (*scope* (i)) will host an MQTT broker.

[0423] (Sparse brokers) Some of the existing CPUs host MQTT brokers, and some do not.

[0424] The specification of the set of MQTT brokers supporting the distributed execution of an application is defined as follows. Given an application specification that includes the ontology E ,

[0425] a clustering specification, a CPU specification, and a placement specification, a communication specification is defined as a pair: the communication hubs : a subset of the active entities whose CPU instances will host MQTT Brokers; and a mapping *broker* () between each pair of communicating clusters $(C1, C2)$ to a communbic hub e where: the entity $e \in \text{ascendants}(\text{lowest_ancestor}(\text{placement}(C1), \text{placement}(C2)))$, the link between *placement* ($C1$) and e is marked as a network edge in the CPU specification, and the link between *placement* ($C2$) and e is marked as a network edge in the CPU specification.

[0426] The syntax to describe the active entities to also specify if that entity is a communication hub is extended as follows.

```
-active-entity:
    entity: building
    communication-hub: true
-active-entity:
    entity: door
-network-communication:
    entity1: building
    entity2: door
```

[0427] In addition, the syntax 3300 to describe a cluster communication specification is shown in Fig. 33. Fig. 34 shows a broker communication 3400 between placed clusters in the running example.

[0428] Semantics. At runtime, given an instance graph I , two things are true: for every active entity e that is the result of a *broker* mapping and for every instance $i \in \text{instances}(e)$ there exists an broker hosted at the CPU(i), identified as $MQTT_broker(i)$, and for any task instance $(T1, i1)$ belonging to a cluster instance $(C1, i2)$ and any task instance $(T2, i3)$ belonging to a cluster instance $(C2, i4)$, then if the two task instances $(T1, i1)$ and $(T2, i3)$ communicate, they will communicate via the broker $MQTT_broker(i0)$ where $i0 = \text{ascendants}(i1) \cap \text{ascendants}(i2) \cap \text{instances}(\text{broker}(C1, C2))$.

[0429] Fig. 35 shows a broker communication 3500 including two task-instances in the running example. For any pair of communicating task instances $(T1, i1)$ and $(T2, i2)$ defined as above, the broker $MQTT_broker(i0)$, that supports the communication, exists and is uniquely identified.

[0430] Distributed Execution Plan

[0431] A complete distribution plan is now described. Given an application specification a distributed execution plan is an internally consistent the set of: a clustering specification, a CPU, specification, a placement specification, and a communication specification.

[0432] All distribution executions plans maintain the same semantics of the application. The decisions concerning the distribution of the processing should absolutely not change the overall behavior of the application. Note that this does not imply that the exact messages will be processed by each task instance node in the runtime graph, or in the exact same order. However,

the distribution results in an execution that is legal and allowed by the semantics of an application.

[0433] Each distributed execution plan has different consequences on various interesting dimensions, among which there are several important ones: financial cost (e.g. initial cost and operational cost), latency from sensors to actuators, bandwidth of network communications, availability and reliability, and the security profile of the system.

[0434] Each application has different constraints on some of those dimensions (e.g. reaction between sensing and action on an actuator needs be under a certain time limit, certain actuators cannot be unavailable less the a certain amount of time per month, the overall cost of the project for 2 years cannot exceed a certain amount). Among the distribution plans that are possible within the application's constraints, business users might have different preferences in terms of their weight on the overall decision (e.g., an agricultural application might prioritize financial cost over 99% availability, while a nuclear plant certainly will prioritize availability and security over cost).

[0435] Distributed execution plans as defined in this paper are static, fixed, and instance agnostic. This statement means that the execution plans are independent of the particular instances the application will be using at runtime, that the plan is established before any part of the execution is in flight, and that the plan does not change during the execution. Choosing the optimal execution plan might take into account statistical knowledge about the instance graph, but not the graph itself.

[0436] The distributed execution plan is obtained by various methods. One method is for the engineer to directly provide the four specifications. Even if manually produced, having a high level specification and having code automatically generated, rather than producing low level code by hand, has an enormous advantages in terms of productivity, automatization and correctness. Engineers can experiment with various execution plans in a very short amount of time and compare them in terms of their desired properties.

[0437] The Running Example Distributed Execution Plan. The running example of distributed execution plan consists of four specifications: the clustering specification (Figure 27), the CPU specification (Figure 29), the placement specification (Figure 31), and the communication specification (Figure 33).

[0438] Fig. 36 shows a distributed execution plan 3600 of the runtime example. Cameras are located at doors (cluster_one). The images are sent to the cloud (cluster_two) for all remaining processing. The running example produces a distributed plan that places almost all computation in the cloud. Data captured by the camera is sent to the cloud, where the remaining processing

occurs. The principle advantages of this execution plan are simplicity and the ability to easily scale-out by using auto-scaling features of cloud vendors. The principle disadvantage of the plan is the network bandwidth cost, since every image is transmitted to the cloud, 24 x 7. Conceptually, this plan is instantiated with a specific set of instances by the customization tool, to produce an instantiated distributed execution plan. Fig. 37 shows a conceptualization of the instantiated distributed execution plan of the runtime example. The actual runtime plan includes additional injected system functionality.

[0439] The main principle of edge computing advises pushing computation nearer to the sensors. Applying this idea the crop function is moved to be co-located with each camera. To generate and study this revised distributed plan, the infrastructure engineer would modify two lines in the cluster specification. No other changes are necessary to any code or other configuration files.

[0440] The new cluster specification containing the two lines of changes for the revised distributed execution plan is as follows.

```
-cluster:
    name: cluster_one
    tasks: [heartbeat, camera, crop]
-cluster:
    name: cluster_two
    tasks: [recognize, enrich, monitor]
```

[0441] Conceptually, the two line modification produces a distributed plan that crops images to contain only faces before the images are sent to the cloud for further processing (Figure 38). The new instantiated plan is automatically produced by the system (Figure 39).

[0442] Comparing the two distributed execution plans, they produce the same output, so the semantics of the application has not changed. Note that in the case that the room is empty, the crop task produces no output, so in the modified plan, no bandwidth is used. Informally, expect at a minimum a $1 - 8/24 = 2/3$ reduction in bandwidth, assuming each room is occupied 8 hours out of 24 hours a day. This computation does not account for the added savings from the reduction in size of cropped images transmitted in the modified plan compared to full images transmitted in the cloud-based plan.

[0443] With respect to latency, the cloud-based plan has higher bandwidth cost and higher transmissions times, but still is expected to have lower latency, since the CPU attached to a cloud cluster typically has a GPU that executes machine-learning models approximately 10x

faster than a CPU. A processor attached to a door typically does not have a GPU because of the additional costs involved.

[0444] Fig. 38. shows a distributed execution plan 3800 of the modified runtime example. Cameras send captured images to a crop task located in the same cluster-instance. The crop task outputs cropped images containing only faces. The cropped face images are sent to the cloud for further processing

[0445] Fig. 39 shows a conceptualization 3900 of the modified instantiated distributed execution plan of the runtime example 3800. The actual runtime plan includes additional injected system functionality.

[0446] Manually investigating possible distributed plans can be time consuming for the case of large applications. Another solution is to use an optimization algorithm to choose among the various plans that satisfy the constraints of design along different "cost" dimensions. In practice, it is very likely that "best" strategy to obtain the "optimal" plan is carefully chosen by a combination of manual and automatic optimization work, where the optimization algorithms suggest plans and are given feedback by humans (system engineers, domain specialists and business decision makers) in a design loop.

[0447] Independent of how the distributed plan is generated, the execution proceeds in the same way. Given a complete execution plan, the Autocoder system will proceed as subsequently described.

[0448] Physical Distributed Execution Plans

[0449] The distributed execution plan explained above refers to "logical" CPUs and "logical" network connectivity. At a certain point during the design of the distributed execution of an application, precise details must be given about which OS is running on the instances of each active instance, which the broker (and its details and configuration) is running on communication hub instances, and which particular network protocol is used on each logical network communication link. This information is needed for implementing a transparent distributed communication, as well for the build and install tools. This decision obeys the same rule of homogeneity that we applied to all our decisions, i.e. all instances of the same entity will run the same OS (if active), they will run the same broker instance, with same exact configuration (if communication hub), and the actual physical network protocols between each pair of instances that belong to the same pair of entities is precisely the same one. Lower-level decisions will impact feasibility and the cost metrics associated with the application: cost, reliability, performance, energy consumption, etc. These issues are considered in a holistic optimization strategy.

[0450] Platform Configuration

[0451] To generate the correct containers, executable in their respective target environments, the data processing system requires information about the OS of the target environments where each piece of code will be executed. This information is described as bellow. Each active entity declares its platform, as in the following example.

```
-active-entity:
    entity: building
    platform:
        OS: aws-s3
-active-entity:
    entity: door
    platform:
        OS: linux-arm64
-network-communication:
    entity1: building
    entity2: door
```

[0452] Broker Configuration. In addition, each communication hub declares the type of MQTT Broker running on instances of that hub and the details of its configuration. This information is added to the active-entity specification, as follows:

```
-active-entity:
  entity: building
  platform:
    OS: aws-s3
  broker:
    type: mosquitto
    configuration-file: building.mosquitto.conf
    port: 9001
-active-entity:
  entity: door
  platform:
    OS: linux-arm64
-network-communication:
  entity1: building
  entity2: door
```

[0453] This information is used for distributed communications, as well for controlling the MQTT brokers automatically (start, stop, check).

[0454] Network Configuration. Each network communication link specifies the particular network protocol implementing it (e.g. Low power Wireless Personal Area Networks (6LoWPAN), Bluetooth Low Energy (BLE)). This information is added to the active-entity specification, as follows.

```
-active-entity:
  entity:
    building
    platform:
      OS: aws-s3
  broker:
    type: mosquitto
    configuration-file: building.mosquitto.conf
    port: 9001
-active-entity:
  entity: door
  platform:
```

```
OS: linux-arm64
-network-communication:
  entity1: building
  entity2: door
  protocol: wifi-4G
```

[0455] Given that most implementations of publish/subscribe protocols (the protocol we use as a fundamental component for distributed communications in our system) hide the underlying network protocol, this choice does not impact any of our software tools. However, the choice of any network protocol will impact the cost of a particular distributed solution and impose limitation on some metrics such as bandwidth and/or energy availability.

[0456] Extensions for Distributed Communications

[0457] One major change in a distributed execution is focused on making sure that messages exchanged between task instances across various different clusters, and potentially on different CPUs, reproduce the logical communication of a centralized execution. This section details how Autocoder achieves this effect. Autocoder injects two additional operators that deal with sending and receiving messages across clusters, and modifying the dataflow graph to explicitly incorporate those two operations.

[0458] Recall that the goal of the dataflow tool is to rewrite the dataflow graph to add various functionality, while maintaining the same basic semantics of the application. This section describes the extension of the centralized dataflow tool for the purpose of integrating distributed communications into the dataflow graph.

[0459] When two task-instances are directly connected and in the same cluster, the task-instances are co-located in the same process, so notifications are implemented with a function call. All communication between task instances that are part of different cluster instances are executed via an intermediary broker.

[0460] Fig. 40 shows a distributed execution plan 4000 of the modified runtime example with adaptors. Cameras send captured images to a crop task located in the same cluster-instance. The crop task outputs cropped images containing only faces. The cropped face images are sent to the cloud for further processing.

[0461] Fig. 41 shows a small portion 4100 of the centralized dataflow graph featuring two connected tasks. The output port of the crop task is named oneface. The input port of the recognize task is named face. The arrow indicates the notification message from the crop task to the recognize task.

[0462] Fig. 42 shows a result 4200 of editing the runtime graph of Figure 41 to introduce adaptors for distributed communication. The arrows indicate the notification messages going to the sending adaptor task and coming from the receiving adaptor task. Communication between adaptor tasks is accomplished through a publish/subscribe system. The task names of the adaptors are automatically generated by the system.

[0463] To implement this communication, the dataflow tool edits the dataflow graph and inserts two tasks that implement the start and end of the distributed communication (Figure 38). The inserted tasks are then used to replace the existing connection (Figures 41 and 42). Formally, for every inter-cluster connection $(T1, port1) \rightarrow (T2, port2)$ that links tasks $T1$ from cluster $C1$ to task $T2$ from cluster $C2$ where $C1 \neq C2$, the follows steps are performed: Add a new adaptor task $A1$, implemented by operator *TaskToClient*, with the same scope as $T1$. This step requires constructing a unique new adaptor name. Add a new adaptor task $A2$, implemented by operator *ClientToTask*, with same scope as $T2$, also with a unique name. Add adaptor $A1$ to cluster $C1$. Add adaptor $A2$ to cluster $C2$. Add a new connection $(T1, port1) \rightarrow (A1, input)$. Add a new connection $(A2, output) \rightarrow (T2, port2)$. Remove the original connection from the original dataflow.

[0464] In addition, instance specific parameters will be given to each instance of a sending or receiving adaptor. Both tasks have an instance-level parameter broker that defines the broker instance used for communication. In addition, both sending and receiving adaptor instances have additional parameters whose value will drive runtime publish and subscription topics as subsequently described.

[0465] Adaptor Operators. The two operators implementing the distributed communication are described in this section: *TaskToClientAdaptor* operator implements sending messages (the sending adaptor), and *ClientToTaskAdaptor* operator implements receiving messages (the receiving adaptor). Each instance of those two adaptor operators will have an internal broker client object, that is initialized and connected to the right broker at adaptor object creation time.

[0466] *TaskToClientAdaptor* Operator. The operator is part of Autocoder's built-in library of operators. The operator description is given bellow.

```
operator:
  name: TaskToClientAdaptor
  input_ports:
    process:
      name: input
```

```

    semantics:
        has__internal__state : false
        mobile: true
    implementation:
        language: python
        strategy: passive
        module: autocoder.tasks.adaptor_task

```

[0467] This operator publishes each input message received on the input port to the given broker. The operator's code uses the given parameters to connect to the correct broker (Section 5.6) and construct the correct publish topics.

[0468] The implementation of the operator consists of the following logic. First, initialize the internal broker client object and connect it to the broker according to the values of then broker parameter. Second, in an endless loop, process message *m* on port input: create the publish topic; serialize the message *m* according to the serialization logic (see A) associated to the datatype of the payload of *m*; and publish the serialized format of the payload using the created publish topic via the internal broker client.

[0469] ClientToTaskAdaptor Operator. The operator is part of Autocoder's built-in library of operators. The operator description is as follows.

```

-operator:
    name: ClientToTaskAdaptor
    output_ports:
        send:
            name: output
    semantics:
        has__internal__state : false
        mobile: true
    implementation: language:
        python strategy: active
        module: autocoder.tasks.adaptor_task

```

[0470] The operator's code uses its given instance parameters to connect to the correct broker and to construct the correct subscribe topics.

[0471] The *ClientToTaskAdaptor* operator is active, so it automatically receives one (or more) separate threads as part of the active operator semantics. Also, like all other active operators, the *ClientToTaskAdaptor* has an internal queue accessed by its various threads.

[0472] During initialization the operator subscribes to the given broker with two subscriptions, one for broadcast messages and one for targeted message. When a topic is published to the broker, the broker attempts to match the topic to the subscriptions. If there is a match, a callback function is invoked in the receiving operator. The callback function deserializes the received messages and queues them in the operator. One of the available threads reads the queue and sends the message to the output port output. First, initialization: create internal broker client and connect to broker; subscribe to messages with callback with topics specified as parameters; start threads; wait on queue. Second, callback on subscribe match: Deserialize received message into the expected Payload data type; (under lock) Add a new event to the queue; (endless loop) Each thread executes the following logic: Wait on Queue until an event is present; (under lock) Remove event from Queue; Send event to output port output.

[0473] Matching Publish with Subscribe. In order to ensure that distributed communication mimics perfectly the centralized communication, we need to carefully craft publishing and subscription topics for each instance of an adaptor operator. The basis of our solution is a strict grammatical structure for our instance identifiers; we will exploit this structure in the publish and subscribe topics.

[0474] Instance Identifiers Structure. Each instance i of an entity e has the identifier:

[0475] $identifier(i) = prefix + entity_name + "/" + local_identifier + "/"$ where the *prefix* of i is (recursively) the $identifier(parent(i)) + "/"$ and $""$ in the case of the root *entity_name* is the name of the entity e the instance i belongs to, and *local_identifier* is a string that uniquely identifies the instance i among the set *instances* (e).

[0476] In the running example one of the doors is identified by building/1/room/2/door/2/, meaning that this is the door "2" of the room "2" of the building "1". Note that identifiers with this grammatical structure are used through the entire paper as examples; however, only the matching between publish and subscribe topics will depend on the structure. All the other techniques described in this paper are agnostic to the structure of instance identifiers, and those techniques only rely on each instance identifier being unique.

[0477] This particular identification mechanism has the following property for use in the distributed execution: all the identifiers of the descendants of an instance i satisfy the regular expression; $identifier(i)*$ where $*$ matches zero or more alpha-numeric characters.

[0478] This property is used to construct publish/subscribe topics that guarantee that messages reach correct targets and no incorrect messages get ever processed. The rest of the discussion of this section does not distinguish between an instance *i* and its unique *identifier* (*i*).

[0479] Subscription and Publishing topics. The newly introduced adaptors use instance-level parameter to generate publish and subscribe topics.

[0480] Sending - Publishing

[0481] An instance *i* of a sending adaptor corresponding to the communication link (from_task, from_port) → (to_task, to_port) will receive the following instance level parameters: broadcast_publish_topic, where broadcast_publish_topic= from_task + ":" + from_port + "/" + *i* targeted_publish_topic_stub, where targeted_publish_topic_stub= "targeted" + ":" + to_task + ":" + to_port + "/".

[0482] In the example, for the dataflow communication link (crop, oneface) → (recognize, face) the values of the topics-related parameters of the sending adaptor adaptor0 are as follows.

-task-instance-parameters:

task_name: adaptor0

instance: building/1/room/1/door/1/

parameters:

-name: broadcast_publish_topic

value: crop:oneface/building/1/room/1/door/1/

-name: targeted_publish_topic_stub

value: targeted:recognize:face/

-task-instance-parameters:

task_name: adaptor0

instance: building/1/room/2/door/1/

parameters:

-name: broadcast_publish_topic

value: crop:oneface/building/1/room/2/door/1/

-name: targeted_publish_topic_stub

value: targeted:recognize:face/

-task-instance-parameters:

task_name: adaptor0

instance: building/1/room/2/door/2/

parameters:


```

-name: broadcast_publish_topic
value: crop:oneface/building/2/room/1/door/2/
-name: targeted_publish_topic_stub value: targeted:recognize:face/
value: targeted:recognize:face/

```

[0483] A sender adaptor publishes a broadcast message with the topic `broadcast_publish_topic` (the given parameter). A targeted message is instead published with a topic created based on the stub and the target identifier as follows: `targeted_publish_topic=targeted_publish_topic_stub+i`, where i is the target instance identifier of that message. In our example, all messages sent by the instance `building/1/room/2/door/2/` are broadcast messages. A message sent by the same instance `11`, but targeted towards the instance `building/1/room/2/` will be published with the topic `targeted:recognize:face/building/1/room/2`

[0484] Receiving - Subscribing

[0485] An instance i of a receiving adaptor corresponding to the communication link $(\text{from_task}, \text{from_port}) \rightarrow (\text{to_task}, \text{to_port})$ will receive the following instance level parameters:

1) `broadcast_subscribe_topic`, where `broadcast_subscribe_topic=from_task + ":" + from_task_port + "/" + match_expression`, where `match_expression` is defined as:

```

if scope(from_task) ∈ descendants(scope(to_task)),
then: match_expression=ascendant(i,scope(to_task))+"#"
else: match_expression=i+"#"

```

[0486] `targeted_subscribe_topic`, where `targeted_subscribe_topic="targeted" + ":" + to_task + ":" + to_port + "/" + i` In this example the values are specified as parameters as follows:

```

-task-instance-parameters:
task-name: adaptor1
instance: building/1/room/1/
parameters:
-name: broadcast_subscribe_topic
value: crop:oneface/building/1/room/1/#
-name: targeted_subscribe_topic
value: targeted:recognize:face/building/1/room/1/
-task-instance-parameters:

```

```

task-name: adaptor1
instance: building/1/room/2/
parameters:
  -name: broadcast_subscribe_topic
  value: crop:oneface/building/1/room/2/#
  -name: targeted_subscribe_topic
  value: targeted:recognize:face/building/1/room/2/

```

[0487] At runtime, an instance of a receiving adaptor will subscribe to the broker with those two particular topics given as parameters. In this example, the instance building/1/room/2/ of the adaptor1 task subscribes to topic crop:face/building/1/room/2/# and topic targeted:recognize:face/building/1/room/2/. The first subscription exploits # wildcard matching of path expressions provided by the MQTT specification.¹² The wildcard will allow matching for example publishing topics crop:face/building/1/room/2/door/1/ and crop:face/building/1/room/2/door/2/. In other words, each room will only receive messages sent by its own doors, which is the expected semantics.

[0488] Adding Broker Information to Adaptors

[0489] An additional parameter of each instance of an adaptor operator is concerning the particular MQTT broker the internal client will connect to. In order to calculate those parameters, our system will need additional information: the IP addresses of each communication hub instance in the system. This information is also provided to the Autocoder, similar to the other specifications.

[0490] **IP addresses for Communication Hub Instances.** For every instance i of a communication hub e , the IP of the CPU(i) (i.e. the CPU logically associated with that instance) must be declared.

[0491] IP addresses can be specified in various ways

```

hub-address:
  instance: building/1/
  ip: server-127.dev.fort-alto.com

```

[0492] The values for the "ip" attribute can be a dotted IP address, a host name that resolve to an IP address with the associated name service, or in the case of cloud computing, "ip" is the identifier of the cloud object provided by the cloud service. Depending on the networking software, additional information can be added here for the connection.

[0493] Adding parameters to adaptors to specify the broker to connect to. . In addition to the parameters corresponding to publish and subscription topics, each instance of an adaptor (sending or receiving) gets an additional parameter whose value is used to connect to the correct broker. An instance i of a sending adaptor corresponding to the communication link $(\text{from_task}, \text{from_port}) \rightarrow (\text{to_task}, \text{to_port})$ where from_task belongs to the cluster from_cluster and to_task belongs to the cluster to_cluster will receive the parameter `broker` composed of two fields: `ip`, obtained from the cluster-communication declaration and the `ip` declaration as follows: `ip(broker)=ip(ascendant(i,broker_scope))`; and `port`, obtained from the `broker` declaration associated with the platform declaration as follows: `port(broker)=port(broker(platform_scope(broker_scope)))`

[0494] where `broker_scope=cluster-communication(from_cluster, to_cluster)` and `platform_scope` returns the scope of the entity argument.

[0495] In our example the values are specified as parameters as follows. The three instances of the sender adaptor will receive the parameters:

-task-instance-parameters:

task__name: adaptor0

instance: building/1/room/1/door/1/

parameters:

-name: broker

value:

ip: server-127.dev.fort-alto.com

port: 9001

-task-instance-parameters:

task__name: adaptor0

instance: building/1/room/2/door/1/

parameters:

-name: broker

value:

ip: server-127.dev.fort-alto.com

port: 9001

-task-instance-parameters:

task__name: adaptor0

instance: building/1/room/2/door/2/

```

parameters:
  -name: broker
  value:
    ip: server-127.dev.fort-alto.com
    port: 9001

```

while the two instances of the receiver parameter adaptor1 have the parameters

```
-task-instance-parameters:
```

```
task_name: adaptor1
```

```
instance: building/1/room/1/
```

```

parameters:
  -name: broker
  value:
    ip: server-127.dev.fort-alto.com
    port: 9001

```

```
-task-instance-parameters:
```

```
task_name: adaptor1
```

```
instance: building/1/room/1/
```

```

parameters:
  -name: broker
  value:
    ip: server-127.dev.fort-alto.com
    port: 9001

```

[0496] Correctness of Distributed Communication. The adaptors and distribution mechanism guarantee that the implementation does not change the semantics of the application. That is, the implementation of publish/subscribe, and associated code, follows the semantics of the application specification.

[0497] The distributed execution of a runtime graph is identical to a centralized execution of the same runtime graph. In other words: no message is received that was not intended for processing, and assuming that the underlying infrastructure (e.g. brokers, network communication) ensures no loss of messages, every message sent in a distributed fashion reaches its intended destinations.

[0498] Adding distribution changes the timing of notifications, but these changes do not change the underlying semantics of the application. Some applications require more rigorous

distributed communication that guarantee properties of messages transmission (such as exactly once message delivery). As discussed previously, such requirements need to be specified explicitly and the distributed implementation relegates them to the underlying publish/subscribe service.

[0499] Distributed Version of Autocoder

[0500] A version of the Autocoder tools are for the purpose of distributed execution of an application. An extended version of the verification tool concerns the verification of the internal consistency of the distributed plan. Some example of correctness rules that are being checked at this stage are as follows: every task in the dataflow belongs to one and only one cluster; every cluster has a specified placement, and that placement is legal (i.e. is an ascendant of the scope of the cluster, and the placement entity is active); and for every pair of clusters that need to communicate there is a declared broker entity and that entity is legal (aka, that entity is a communication hub, there are network links between the placements of the two clusters and the communication broker entity).

[0501] In addition, in order to proceed with the distributed execution, the data processing system checks the information about the OS of the CPUs of each active entity and brokers that are used on each communication hubs: for each active entity e , there is an OS platform declaration; and for each communication hub e , there is a declaration of the broker at that hub. In addition, the validator checks for appropriate information about the IP address of each instance of a communication hub: for each communication hub e and every instance $i \in \text{instances}(e)$, there is a declaration of the ip address of the CPU(i).

[0502] A dataflow compiler tool simply injects the logic concerning the distributed communication. In addition, optionally, the dataflow compiler can inject functionality with respect to global control.

[0503] Build Tool. Building a runtime consists four different entities - the development platform, the build environment, the build scripts, and the execute platform. The build system constructs the build environment on the development platform, for example MacOS. The build environment is then copied to a replica of an execute platform, for example, Raspberry Pi Linux Arm64. The build script is invoked, resulting in an executable container. The containers thus produced are ready for deployment.

[0504] The Autocoder provides a default build environment based on secure shell functionality. For the distributed case, the build steps, executed on the development platform, are as follows: assemble build environments for each cluster; generate build scripts for each

cluster; copy the build environments and scripts to the target build platforms for each cluster; and run build scripts on target execute platforms. This process compiles code, generates containers, etc.

[0505] Note that the set of all cluster instances (that will each become a container) running in the distributed system is: $\{(C, i) \mid C \text{ is a cluster and } i \in \text{instances}(\text{scope}(C))\}$.

[0506] The content of each build environment corresponding to a cluster instance (C, i) is composed of the following: the anchored dataflow specification, projected only to the set of tasks in the cluster C ; the instance graph, projected only to the *descendants* (i), the code of all operators corresponding to tasks in C , together with their libraries specified in the pip description, the application parameters, projected as follows: the parameters of all operators of tasks in the cluster C , the parameters of all tasks in C , the parameters of all task instances (T, i') where the task T is in the cluster C , and $i' \in \text{descendants}(i)$; all the data (static files) referred to by parameters in the list above; the runtime environment; and system libraries used in the rewritten dataflow (projected on the cluster C).

[0507] Embedded Low Resource Devices

[0508] The Autocoder runtime is implemented with a small footprint, since the runtime is intended to run on most devices. Nevertheless, some embedded devices are resource poor and cannot accommodate the runtime. This case is modeled as part of the platform specification. If the platform cannot accommodate the runtime (or uses its own runtime), then the build step adds the platform specific runtime. Typically these platform runtime systems integrate with Autocoder in two ways. In the simplest method, the runtime integrates with the publish/subscribe backend system directly. The platform has its own adaptor implementation specifically designed for integration. For platforms where the computational resources are so limited that adaptors are not possible, a proxy system is generally used. The proxy has the resources to run the Autocoder runtime and serves as a hub to the low-resource devices.

[0509] Execute Platform. The build environment for each cluster instance (C, i) is transformed into a container on a platform specified by the *os* ($\text{platform}(\text{placement}(C))$): each container C is to be executed on a CPU located in an instance of *placement* (C), and the operating system running on that CPU is *os* ($\text{platform}(\text{placement}(C))$).

[0510] Build scripts. A system library of build scripts for each of the common OS platforms is provided.

[0511] Install Tool. The result of the build is a set of containers. Each container (C, i) is executed on the CPU associated with the instance *ascendant* ($i, \text{placement}(C)$). The final

steps are simple: Copy build results and install scripts corresponding to each container (C, i) on its intended target execution platform *ascendant* ($i, placement(C)$). Run install scripts to install clusters.

[0512] Data Types

[0513] The data type system consists of a loose coupling between the types and the code. Types are defined as code in the given programming language. The following requirements hold for data types: (1) A data type is a base type of programming language, or a class. (2) A class inherits from a well defined class that provides default implementations of some services. (3) The class serializes relevant information with an encoding method. (4) The class deserializes with a decoding method. (5) If a class instance is rendered in a display format, the class implements the rendering method.

[0514] For example, the Image type implementation in the Python system library has a class that contains the data. The class inherits from the Payload system class. The class has a constructor implementation that takes various input formats to construct an image object. The encode method returns the image encoded as a JSON object with metadata and base 64 encoded image data. The decode method constructs the image from the result of the encode method. The to_html method returns an HTML anchor that embeds the image as a data-encoded URL.

[0515] Data types most frequently appear as additional annotations for the ports of an operator. Annotations appear in the schema section of the implementation part of the operator declaration. A port may declare multiple data types.

```
- operator:
  name: CameraSensorPiCamera
  input-ports:
    - process:
      name: heartbeat
  output-ports:
    - send:
      name: image
  implementation:
    language: python
    module: demo.tasks.camera__sensor__pi__camera
    schema:
      input-ports:
        name: heartbeat
        type: int
      output-ports:
        name: image
```

```

        type: Image
    pip:
        - "picamera==1.13"
        - "Pillow==8.3.2"

```

[0516] Fig. 43 shows specification and implementation details of the CameraSensor operator 4300. The schema declaration in the implementation defines the schema associated with each message type. The default implementations of encode simply recursively encodes any local variables in the class. If the value of the variable is not encode-able, the encoding fails with an error. The default implementation of decode constructs an object assuming the structure of the output of the default encode method. The default implementation of the to_html method renders the object data as an HTML element.

[0517] Operator Templates and Inheritance

[0518] Operator templates are superclasses that are inherited by user operator code. The operator declaration includes an "extends" annotation as part of the implementation part of the operator declaration. The value or values of the annotation are the name of a user or system class, a dot notation path to a class, or a list of such objects. The automatically generated superclass of an operator includes references to these classes. Thus, by inheritance, the operator code inherits the superclasses.

[0519] Grouping Tasks into Taskviews

[0520] Task views provide an encapsulation method for tasks. A task view is defined as an interface to a set of tasks, connections and parameters.

[0521] Let T be a set of anchored tasks. Let C be the set of connections that connect two tasks in T . Let O be the set of operators of the tasks in T . Let IP be the set of (input port, operator) pairs of the operators of T and OP be the set of (output port, operator) pairs of the operators of T . Let PT be the set of task parameters and PO be the set of operator parameters of T .

[0522] Then a task view is defined with the following information: (1) A task with an identifier V . (2) A set of input ports VI for the task view. (3) A set of output ports VO for the task view. (4) A one-to-one mapping from VI to a subset of I . (5) A one-to-one mapping from a subset of O to VO . (6) A set of task parameters VP for the task view. (7) A mapping MT from VP to the task parameters PT or operators parameters PO . The mapping must cover all the parameters. (8) A mapping from a set of anchors A to the set of tasks V .

[0523] From the application semantics point of view, the view V is "compiled out" by replacing the view with its contents. That is, for an anchored dataflow graph G with task view

V, (1) Insert T into G, replacing the anchors in T with those of A. (2) Insert C into G. (3) For a connection c of G that connects to an input $V I$, connect c to the mapped input port. (4) For a connection c of G that connects to an output port in $V O$, connect c to the mapped output port. (5) Replace every parameter in $V P$ with the corresponding parameter designed by MT . Note that if an anchor of the task T refers to the entity ontology of G , the anchor set A need not contain that anchor.

[0524] Library of Predefined Generic Operators

[0525] The Autocoder system contains a library of predefined operators for common patterns that occur in dataflow systems. The use of these operators decreases the amount of code written by the software engineer and increase code quality.

[0526] Every operator requires some additional information. An operator in this library is either a regular operators or a template operator. A regular operator is generally configured by providing some parameters (to the operator, task, or instance). The software engineer provides required additional code and parameters to a extension operator. A template operator has parameters but also a superclass that the user operator inherits.

[0527] format for each operator: 1. input ports, output ports. 2. mobile? stateless? active? passive? 3. Parameters. 4. required methods. 5. template code. 6. Functionality. 7. Validation

[0528] Automata Template

[0529] The Automata Template provides a generic interface to a finite state automata. A finite automata is defined by a set of state symbols, an input alphabet, an output alphabet, an initial state, and a transition matrix. The transition matrix produces a new state given an existing state and an input value.

[0530] Table 11: The Automata Template properties.

Feature	Value
class name	AutomataTemplate
input port	<i>input</i>
output port	<i>output</i>
mobile	true
stateless	true
logic	passive
parameters	<i>initial_state, transition_matrix</i>
methods	<i>classify : event $\rightarrow$ state symbol</i> <i>$x_i : state_symbol \rightarrow void$, one for each state symbol x_i</i>

[0531] The template receives an event on an *input* port and sends its output to any number of output ports as defined by the extension operator. The automata is mobile, has state, and is executed by default passively.

[0532] The automata structure is defined by two parameters, *initial_state*, and *transition_matrix*. The initial state parameter value is a state. The transition matrix is a dictionary with keys that are states. Each key has a value that is another dictionary that maps an input alphabet symbol to the new state.

- name: *initial_state*
value: *state_apple*
- name: *transition_matrix*
value: {*state_apple*: {0: *state_apple*, 1: *state_orange*, 2: *state_orange*},
state_orange: {0: *state_apple*, 1: *state_apple*, 2: *state_apple*}}

[0533] Fig. 44 shows an example parameter set 4400 for an automata that uses the AutomataTemplate. At object creation time, the initial state parameter must exist and its value must occur in the transition matrix; otherwise, an error is raised.

[0534] The extension code consist of two parts. The first part is a single method *classify* $y: event \rightarrow input$. This method is called when the task-instance receives an event. The output of the event is an automata input alphabet symbol.

[0535] The second part consists of a set of methods, one per state, of the form *state_name*: $event \times old_state \times input \times new_state \rightarrow void$

[0536] When the automata transitions to *state_name*, the corresponding method is invoked. These methods are typically used to send messages (symbols in the output alphabet).

[0537] At object creation time, a method must exist for every state, otherwise an error is raised.

```
class AutomataTestTask(AutoAutomataTestTask):
    def classify(self, event):
        n=event.get_payload()
        return (event.get_payload()) %3
    def state_apple(self, event, old_state, new_input, new_state):
        self.write_apple([Event(str(event.get_payload()))])
    def state_orange(self, event, old_state, new_input, new_state):
        self.write_apple([Event(str(event.get_payload()))])
```

[0538] Fig. 45 shows an example Python classify and state methods 4500 of the example automata that uses the AutomataTemplate. After initialization, the operator simply waits for

an input event. The event is translated into an input alphabet symbol via the classify method, and then the appropriate transition function is invoked.

```
def process_input(self, event):
    self.old_state = self.state
    self.new_input = self.classify(event)
    self.new_state = self.transition_matrix[self.state][self.new_input]
    fun = getattr(self, self.new_state)
    fun(event, self.old_state, self.new_input, self.new_state)
    self.state = self.new_state
```

[0539] Fig. 46 shows an example sketch 4600 of the transition method of AutomataTemplate in the Python library.

[0540] Heartbeat Operator

[0541] The Heartbeat operator repeatedly generates an empty event, separated by a time delay selected from a given distribution, as given in the parameters.

[0542] Table 12 shows the heartbeat operator properties.

Feature	Value
class name	AutomataTemplate
input port	none
output port	heartbeat
mobile	true
stateless	true
logic	active
parameters	random_seed, function, arguments

[0543] The operator has no input port and an output port named heartbeat. The operator is mobile, stateless, and has an active execution strategy. The operator takes three required parameters. The random_seed parameter is used to initialize the numpy random number generator library in the Python runtime (or some equivalent random number generator in another runtime). The function parameter is the name of a function in the numpy library. The arguments parameter is a dictionary of arguments passed to the function to draw a random number. For example, the normal distribution can be specified with the following parameters.

```
"random_seed": None,
"function": "normal",
"arguments": {
    "loc": 0.0,
    "scale": 1.0
}
```

[0544] The operator executes an infinite loop with two sequential internal steps. (1) Send an empty event to the output port. (2) Pick a random value from the given distribution and wait the chosen value of time in seconds.

[0545] Database Operator

[0546] The Database operator accepts a call with an SQL statement and SQL execution parameters as an argument. The operator then executes the SQL statement against the given database, and returns a complete answer to the calling operator.

[0547] Table 13 shows database operator properties.

Feature	Value
class name	Database
compute port	<i>execute</i>
output port	None
mobile	true
stateless	true
logic	passive
parameters	<i>db_user, db_password, db_host, db_port, db_database</i>

[0548] At operator initialization, the operator establishes a connection to the database using the provided parameters and creates a connection object. The operator has a call port *execute* that takes a Database Request object as a parameter. The object contains two properties, the statement and parameters. The implementation of the call passes these properties to the database system for execution and then returns the result.

```
def call_execute(self, dbrequest):
    with self.connection.cursor() as cursor:
        cursor.execute(dbrequest.query, dbrequest.parameters)
        return cursor.fetchall()
```

[0549] Fig. 47 shows an example sketch 4700 of the Database operator implementation.

[0550] Data Enrichment Template

[0551] Consider a fraud detect pipeline for a credit card charge. In a simplified case, application is designed as a pipeline. The incoming charge request starts the pipeline. Then the application looks up information a series of sources for fraud evidence (past history of the card, a fraud database, credit score, etc.). Evidence is gathered and then presented to a model that renders a score on the probability that the charge is fraudulent. Each "look up" operation is structured similarly. An input record arrives, a key is extracted from the record and submitted to a data source. The data source generates an answer (potentially empty). The answer is merge into the input to create the output, which is sent to the next source. The Data Enrichment Template is designed to support this common analysis pattern.

[0552] Table 14 shows DataEnrichmentTemplate operator properties.

Feature	Value
class name	DataEnrichmentTemplate
input port	input
output port	output
call port	fetch
mobile	true
stateless	true
logic	passive
parameters	None
methods	project : event → key, merge : event × data → void

[0553] The implementation of DataEnrichmentOperator is simple and can be expressed in one line of code. The project method translates the input event to a key. The key is passed to a call to fetch that retrieves data based on the key. The returned information, and the input event, are passed to the merge method. The merge method sends the new merge event to the output port.

```
self.write_output(self.merge(event, self.call_fetch(self.project(event))))
```

[0554] Fig. 48 shows an example sketch 4800 of the Data Enrichment operator implementation.

[0555] Data Generator Operator

[0556] A common pattern in software and system engineering is to develop and test a component in isolation. This methodology requires the generation of synthetic data, typically including errors and anomalies. The DataGenerator operator supports this common pattern. Note that the Heartbeat operator controls the frequency of data event generation, and the DataGenerator operator supports the function that describes the values generated over time.

[0557] Table 15 shows Data Generator operator properties.

Feature	Value
class name	DataGenerator
input port	heartbeat
output port	output
mobile	true
stateless	true
logic	passive
parameters	limit, random_seed, generator

[0558] The DataGenerator operator has a heartbeat input port. For each event that arrives, the distribution is sampled and the resulting value is sent as an event on the output

port. The operator is mobile and stateless. (Stateless in the sense that the distribution is initialized when the class operator initializes an object.)

[0559] The DataGenerator operator is specified through parameters. The *limit* parameter sets a limit on the number of output values (or None for unlimited). The *random_seed* parameter is used to initialize the distributions.

distributions:

```
normal_temperature:
  function: uniform
  arguments:
    low: 36
    high: 37
high_temperature:
  function: uniform
  arguments:
    low: 38
    high: 40
```

[0560] Fig. 49 shows example file generator_distributions.yaml 4900 for parameters. Each *generator* parameter specifies a distribution to generate values. The value of the parameter is a pair: name of the distribution and an object that contains the distribution element (which names the *numpy* distribution). An additional *anomaly* argument specifies another distribution and a *rate*. The rate is the probability that the anomaly distribution is sampled instead of the regular distribution. Note that any arguments for the distribution can be passed to the *numpy* method using the *arguments* element.

```
- parameter:
  name: gentemperature
  type: task
  values:
    - include:
        yaml: generator_distributions.yaml
    - name: limit
      value: null
    - name: random_seed
      value: null
    - name: generator
      value:
        temperature:
          distribution: normal_temperature
          anomaly:
            distribution: high_temperature
```

type: random
rate: 90

[0561] Fig. 50 shows example parameters 5000 for the DataGenerator operator.

[0562] **Event-Condition-Action Template**

[0563] A common data pattern in data flow consists of event-condition-action rule systems. In these systems, when an event arrives, the event is evaluated by a series of conditions. If the condition is true with respect to the event, the associated action is executed. The EventConditionAction template supports this functionality.

[0564] During initialization, the operator reads the parameter for a list of ports. The operator then waits for an event. When an event arrives, the operator loops through the given ports, invoking the associated condition method. If the condition is true, it invokes the associated transformation method, and then writes the generated event to the associated output port.

[0565] Table 16 shows the Event Condition Action Template operator features and properties.

Feature	Value
class name	EventConditionActionTemplate
input port	<i>input</i>
output ports	Provided by the subclass
mobile	true
stateless	true
logic	passive
parameters	<i>event_condition_action_template_output_ports</i>
methods	For each output port <i>x</i> : <i>condition_x</i> : <i>event</i> $\rightarrow$ <i>boolean</i> , and <i>transform_x</i> : <i>event</i> $\rightarrow$ <i>event</i>

```
def process_input(self, event):  
    for port in self.event_condition_action_output_ports:  
        condition_function = getattr(self, "condition_" + port)  
        if condition_function(event):  
            transform_function = getattr(self, "transform_" + port)  
            self.write(port, transform_function(event))
```

[0566] Fig. 51 shows an example sketch 5100 of the main processing of the EventConditionAction template.

[0567] **HTTP Operator**

[0568] Table 17 shows the HTTP operator features and properties.

Feature	Value
class name	HttpRequest
call port	execute
mobile	true
stateless	true
logic	passive
parameters	spec, authentication
methods	For each output port x , $condition_x : event \rightarrow boolean$, and $transform_x : event \rightarrow event$

[0569] A common data pattern in data flow system is the request via HTTP to an external service. This operator supports <https://pypi.org/project/openapi3> to make requests to an external HTTP service based on a OpenAPI specification.

[0570] The operator expects two parameters. The *spec* parameter contains an Open API specification as a python object. The *authentication* parameter contains two elements, the *security_scheme* is a string stating the security scheme, and *value* is of any type, passed to the security scheme.

[0571] When the operator *execute* port is called, an *operation_id* is passed as required by the Open API specification. The operation invokes the Open API specification to call the remote source via HTTP. The result of this invocation is returned to the caller.

[0572] **File Logger Operator**

[0573] Table 18 shows the File Logger operator features and properties.

Feature	Value
class name	FileLogger
input port	input
output port	output
mobile	false
stateless	true
logic	passive
parameters	log_port_type, log_port_name, log_task_name

[0574] A common pattern in computational systems logs in-transit data at different points in an application. The FileLogger operator supports this pattern by writing its input to the local file system of the operator instance. This operator can be manually inserted into the dataflow, or automatically injected.

[0575] The operator utilizes three parameters. The *log_port_type* parameter is typically set to the log port type (*input*, *output*). The *log_port_name* parameter is set to the name of the port being logged. The *log_task_name* parameter is set to the name of the task getting a log on its port.

[0576] At initialization, the operator constructs a log file name with the following components: the log port type, the log port name, the log task name, the instance id

(converted into a safe file representaton by replacing / with), and the file extension ".log". This file is opened in append mode. When an event arrives, the serialization of the event appended to the end of the file. Note that the events contain timestamps.

[0577] **File Replay Operator**

[0578] **Table 19 shows the File Replay operator features and properties.**

Feature	Value
class name	FileReplay
input port	None
output port	<i>output</i>
mobile	false
stateless	false
logic	active
parameters	<i>input_file</i>

[0579] A common pattern in applications is to record a log of the input and then reply the log for debugging or performance measurement reasons. The *FileReplay* operator supports this pattern. At initialization, the operator opens the file named in the *input_file* parameter.

[0580] During normal processing, the operator reads the first event in the file (typically generated by the FileLogger operator) and notes the timestamp of the event. This timestamp becomes the time basis for subsequent events. The operator then sends the event to its output. The operator then reads the next event from the file, computes the difference in timestamps, and sleeps for the time difference. The operator then sends the event. The operator continues in this way, reading the next event, sleeping, then writing the event to output, until the input file is exhausted. Then the operator generates an error to indicate that replay has finished. This algorithm reproduces the timing and data recorded in the log.

[0581] **Streaming Template**

[0582] A common pattern in stream systems aggregates data over a data stream window. Aggregation can be done over sliding or tumbling windows of events, for example. The *StreamingTemplate* supports this common pattern.

[0583] The user declares an operator that inherits from *StreamingTemplate*. During operator initialization, the template code connects the operator input to a instance method *input*. The user operator references this operator to acquire input events. The user operator is required to define a single method, *query* that returns events. These events are sent to the output port. The definition of the query method can be written in Python or PythonQL.

[0584] **Table 20 shows the StreamingTemplate operator features and properties.**

Feature	Value
class name	StreamingTemplate
input port	<i>input</i>
output port	<i>output</i>
mobile	true
stateless	true
logic	passive
parameters	None
methods	<i>query</i>

```
def query(self):
    return [ Event("room full:" + len(w) + "people")
            for tumbling window w in self.input ()
            start x at s when True
            end y at e when (timestamp(y)-timestamp(x)) > self.window_seconds
            where len(w) > self.window_limit ]
```

[0585] Fig. 52 shows an example sketch 5200 of the query method of user code that inherits from StreamingTemplate. The query is written in PythonQL format. Note that the query uses a parameter `window_limit`

[0586] **Library of Operators**

[0587] **CameraSensorPi Operator**

[0588] A common sensor type in IoT systems is a camera. Because of the structure and support for operators in Autocoder, a camera operator can easily be written. The operator takes an input heartbeat message to trigger taking an image. When the images has been read from the device, the image data is sent as an event to the *image* output port (Table 21). The Raspberry Pi version of this operator can be written with 21 lines of Python code.

[0589] **Table 21 shows the CpuController operator features and properties.**

Feature	Value
class name	CameraSensorPi
input port	<i>heartbeat</i>
output port	<i>image</i>
mobile	false
stateless	true
logic	passive
parameters	<i>camera_width, camera_height</i>

```
class CameraSensorPiCamera (AutoCameraSensorPiCamera):
    def __init__(self, *params, **kw_params):
        super().__init__(*params, **kw_params)
        self._camera = None
        self._stream = None
```

```

        self._image = None

    def process_heartbeat(self, _):
        self._stream = BytesIO()
        self._camera.capture(self._stream, format='jpeg')
        self._stream.seek(0)
        self.image = PillowImage.open(self._stream)
        self.write_image([Event(Image(image=self._image))])

    def init_resources(self):
        self._camera = PiCamera()
        self._camera.resolution = (self.camera_width, self.camera_height)
        self._camera.start_preview()
        sleep(2) # camera warm up

    def stop_resources(self):
        self._camera.close()

```

[0590] Fig. 53 shows an example sketch 5300 of the CameraSensorPi operator.

[0591] **Crop Face Image Operator**

[0592] A common operation with a camera is the recognition of faces in an image - that is, computing the bounding box of a face in an image. Face recognition is useful in itself as a filtering mechanism for a library of images. Face recognition is also a step in face identification - the association of a face as a match with a database of individual identities.

[0593] Table 22 shows the CropFace operator features and properties.

Feature	Value
class name	CropFace
input port	<i>image</i>
output port	<i>oneface</i>
mobile	false
stateless	true
logic	passive
parameters	<i>face_limit, face_minimum_size</i>

[0594] When the operator instance initializes, it loads the parameters of its machine learning model into memory. The operator then waits for an event with an image payload to arrive on its *image* input port. The operator then executes the model against the image. This execution results in a (possibly empty) set of bounding boxes on the image. Each bounding box surrounds a face in the image (with high probability). Each face is then cropped from the image and sent to the output port *oneface*. The parameter *face_limit* limits the total number of faces cropped from an image. The parameter *face_minimum_size* requires any cropped image to have at least this parameter number of pixels.

[0595] Functionality Injection

[0596] The application specification offers unique opportunities to extend the functionality of an application. The specification represents the abstract meaning of the application in the form of data. The runtime-graph represents the concrete meaning of the program. The specification can be manipulated in well-defined ways, adding functionality to the specification. Changes to the specification are automatically reflected in the concrete runtime-graph representation.

[0597] This section describes functionality injection, where functionality is added to an application specification by adding or replacing ("injecting") operators, tasks, and connections.

[0598] A pattern for each functionality injection: 1. The new operator. 2. high level specification of the change. 3. the dataflow modification the injection. 4. when during the processing is it happening.

[0599] Global Cluster Controller

[0600] The global cluster controller acts as a centralized hub for managing and controlling the connected devices, sensors, actuators, and subsystems spread across different locations, as represented by task-instance pairs. The global cluster controller provides a unified interface to monitor, configure, and control these entities, allowing for seamless integration and coordination of IoT components.

[0601] The global cluster controller provides a scalable and extensible architecture to integrate new devices and subsystems as the system expands or evolves. The controller itself is written using the underlying architecture, so the controller automatically can support different communication protocols, data formats, and interoperability standards, as they are added to the underlying architecture. This fact allows for seamless integration with existing and future IoT components.

[0602] The global task-instance controller system consists of two operators, the *CpuController* (Table 23) and the *GlobalClusterController* that work in concert to control an application. The *CpuController* operator controls the *ProcessController* operator that controls a cluster on a CPU. The *GlobalClusterController* operator controls the *CpuController* operators running on the CPUs of the application.

[0603] For the *CpuController* The *start_broker* and *stop_broker* input ports causes the operator to essentially issue operating system commands to start and stop a broker from receiving publish and subscribe commands. The *start_clusters* and *stop_clusters* input ports similarly issue commands to start/stop a cluster process. The *force_stop_clusters* input port

forces the clusters to stop processing, regardless of the internal state of the cluster. This command is useful for a cluster in an infinite loop, for example. For error management, the error fast input port is available to report errors from any *ProcessController* to the *CpuController*.

[0604] The *GlobalController* is injected into the system with a unique task name at the root of the entity hierarchy. The *GlobalController* is accessed externally, from an operating system shell, to send commands to it.

[0605] Table 23 shows the *CpuController* operator features and properties.

Feature	Value
class name	<i>CpuController</i>
input ports	<i>start_broker, stop_broker, start_clusters, stop_clusters, force_stop_clusters</i>
fast port	<i>error</i>
output port	<i>shutdown</i>
mobile	false
stateless	false
logic	active
parameters	<i>containers</i>

[0606] Table 24 shows the *GlobalController* operator features and properties.

Feature	Value
class name	<i>GlobalController</i>
input port	<i>error</i>
output ports	<i>start_broker, stop_broker, start_clusters, stop_clusters, force_stop_clusters, shutdown, stop</i>
mobile	false
stateless	false
logic	passive

[0607] At injection time, the follow modifications are made to the specification: For each CPU, create a task *Ti*, with a unique name, scope entity of the CPU, and operator *CpuController*. Insert *Ti* into the specification graph; Create a task *G*, with a unique name, scope of *root*, and operator *GlobalController*. Insert *G* into the specification graph; For each *Ti*, connect the *start_broker, stop_broker* ports to the *ProcessController* for the broker for the CPU; For each *Ti*, connect the *start_clusters, stop_clusters* ports to the *ProcessController* for every cluster on the CPU; For each *Ti*, connect the *shutdown* and *stop* ports to every *ProcessController*; For each *Ti*, connect the error output port of all the *ProcessController* of the CPU to the *error* fast input port of the *CpuController*.

[0608] **Logging**

[0609] A common pattern for development of software systems involves logging, since traces of the operation of a system are useful to understand how a system operates. In addition, population of a database with the outputs of a dataflow application is itself a form of logging. Finally, logging is an important property for the Zoom Debugging functionality (Section 12).

[0610] The *Logger* operator supports this common pattern. The operator is used in conjunction with *FileLogger* and *DatabaseLogger* via multiple inheritance. In effect *Logger* is the base implementation of the operator and *FileLogger* and *DatabaseLogger* providing additional details.

```
- log:
  name: test
  type: input_port
  compute:
    task: checktemperature
    input_port: input
```

[0611] Fig. 54 shows a sketch 5400 of the user logging declaration.

[0612] Debugging Dashboard

[0613] A common pattern in software development involved debugging by monitoring the real time performance of the application. The dashboard declaration of Autocoder supports this activity.

[0614] The developer declares a dashboard through the Autocoder tool (via a text document, GUI, or conversational interface). The declaration causes Autocoder to construct and launch a server that supports a browser dashboard interface GUI. No coding is required from the developer. The interface is automatically constructed from the information available in the specification.

[0615] At a high level, the software architecture for this functionality consists of two main operators. The *WebService* operator runs HTTP services and dispatches HTTP get and post requests from the browser to the *DashboardWebService* operator. The result of the dispatch is returned to the browser. The *DashboardWebService* operator provides two functions: (i) process dispatched requests by returning HTML, and (ii) buffer data arriving from task-instances as declared by the connections in the dashboard declaration. The result is a webpage with an table for each declared connection. For declared connections that receive events, the table contains one row for each task-instance event received (with a timestamp, if provenance information is available). For call events, one call is executed per browser page refresh. The result of the call is added as a new row to the table.

```

- dashboard:
  name: dashboard
  description: test dashboard
  scope: building
  placement: building
  port: 8000
  thread_parallelism: 10
  data:
  - connection:
    from: { task: heartbeatcamera, output_port: heartbeat }
  - compute:
    compute: { task: camera, port: camera_status }
  - connection:
    from: { task: crop, output_port: oneface }
  - connection:
    from: { task: enrichment, output_port: output }

```

[0616] Fig. 55 shows an example of the dashboard declaration 5500.

[0617] At injection time, the follow modifications are made to the specification. (1) A *WebService* operator and associated unique task *W* are inserted at the declared entity scope and placement. (2) Instance parameters *ip*, *port*, *thread_parallelism* are added for the task-instance of *W*. (3) A *DashboardWebService* operator and associated unique task *W* are inserted at the declared entity scope and placement. (4) For each instance of a declared task connection, the task-instance is connected to the *M* to a unique port associated with the connection.

[0618] At initialization, the Webservice operator starts HTTP services connected to the associated IP address of the instance and port as declared. The declaration parameter *thread_parallelism* is converted into an instance parameter. This parameter controls the parallelism of the HTTP services.

[0619] During execution, the *WebService* HTTP service waits for a request. When a request arrives, it is dispatched to the *DashboardWebService* that creates an HTML response, based on any queued data and any real-time calls to instance state, that is returned to the web service, that returns the HTML to the browser.

[0620] During execution, the *DashboardWebservice* waits for arriving events. When an event arrives, the data of the event is added to a queue for the associated port. This queue is managed to maintain a reasonable length. The data in the queue is converted into HTML when a dispatched request arrives.

[0621] The structure of the dashboard system operates on any data contained in the monitored events. If a class of the event payload data contains a *to_html* method, this method is (recursively) invoked when the event is rendered for display. If the method is unavailable, the dashboard system attempts to convert the object to a string via the *str* system method (for the Python implementation). If this latter method fails, the object is rendered as the string "(object)". This set of design choices means that the dashboard always produces something. To improve the quality of the user experience, generally a *to_html* method is declared for all data in the type system.

[0622] More complex dashboards, which leverage more complex web service frameworks (e.g., Flask for Python, J2EE for Java), are straightforward extensions to this system.

[0623] **Adding External Sensors and Actuators**

[0624] A common pattern in Internet of Things is the deployment of devices with a small computational footprint, due to power or cost requirements. Autocoder supports devices with footprints that are too small to support the runtime system.

[0625] The simplest connection consists of devices that connect directly to a broker. In this case, the device publishes/subscribes to a broker in the data serialization format used by the system. The IP address, topic, and a data format (example) are generated by the Autocoder tool. Extracting, transforming, and loading this information into the IoT device is currently the responsibility of the software developer. However, creating a library of supported devices is straightforward.

[0626] **What-If Failure Analysis**

[0627] This section studies the effect of various failures on the overall health of the system. Autocoder provides support for the the management of errors that are generated from underlying faults in the system. In addition in this section we provide algorithms that describe the *consequences* of a failure in terms of the cascading effect of a failure on the set of task-instances. These algorithms can be used in several ways. For example, during the design phase, the algorithms can be used to perform "What-if" analysis to study the potential weak points in a model. During operations, the algorithms can be used to identify the seriousness of an operational failure.

[0628] Our algorithm relies on a binary definition of failure - a component is either operational or not operational. Of course, a particular failure may have a wide range of effects on the semantics of an application. For example, the effects of a dropped incoming messages on an affected task-instance range from one extreme to another. The dropped

message can make the task-instance completely non-operational, or the dropped message can have no effect, or anything in between. In order to decide the level of gravity of the result of failures on the affected task-instances, additional semantic information about the importance of each port of each operator needs to be given. For this section, the analysis globally calculates the set of affected task-instances for a failure, independently of the gravity of the failure.

[0629] There can be several types of failures: *software failures*. Examples of such failures are as follows: task-instances encounter, during their normal processing, errors from which they cannot recover (see [4]). We call such task-instances to be in a *zombie mode*, as they are technically "alive", yet they cannot continue to process incoming messages due to internal state inconsistencies; An entire cluster instance can fail; A broker instance can fail. *hardware failure*. Examples of such failures are as follows: A CPU can become non-operational; A network communication where the communication between two CPUs fails.

[0630] We will discuss each case in particular. For each case we will provide the algorithm that calculates the set of task-instances that are being affected by the given failure. By a task-instance $T(i)$ "being affected" by a certain failure we mean that some of the correct input messages that should have been received by $T(i)$ during normal execution are no longer reaching their target.

[0631] **Task-Instance Failure**

[0632] A task-instance failure occurs when a particular task-instance labeled $T(i)$ becomes non-operational. The set of the other task-instances $T'(i')$ that will miss incoming messages as result of this failure is the set of task-instances such that there exists a path from $T(i) \rightarrow \dots \rightarrow T'(i')$ in the runtime graph. We denote this set as $Failure_{task}(T, i)$ as calculated by the following algorithm. The set of input specifications to this algorithm are: the dataflow specification (includes the set of tasks and their connections), and the instance graph.

Algorithm 1 Failure of a task-instance $T(i)$

```

 $Failure_{task}(T, i) \leftarrow \{T(i)\}$ 
 $\delta \leftarrow Failure_{task}(T, i)$ 
while  $\delta$  non empty do
     $temp \leftarrow \{T_2(i_2) \mid \text{A task } T_1(i_1) \in \delta \text{ such that it exists a task } T_2 \text{ and instance } i_2 \in \text{scope}(T_2) \text{ such that } T_1 \rightarrow T_2 \text{ in the dataflow and } i_1 \rightarrow i_2 \text{ is in the connected instance graph}\}$ 
     $\delta \leftarrow temp \setminus Failure_{task}(T, i)$ 

```

$Failure_{task}(T, i) \leftarrow Failure_{task}(T, i) \cup \delta$

end while

[0633] This algorithm uses the simplified version of the model, where the entity graph and the instance graph are limited to be trees. In fact, the algorithm extends naturally to the non-restricted case of a graph.

[0634] **Cluster Instance Failure**

[0635] A cluster instance failure occurs when a particular cluster instance $C(i)$ becomes non-operational. The set of the task-instances $T_2(i_2)$ that will miss incoming messages are the union of the sets $Failure(T_1, i_1)$ where $T_1(i_1)$ are task-instances in the cluster instance $C(i)$. We denote this set as $Failure_{cluster}(C, i)$ as defined by the following algorithm. The set of input specifications to this algorithm are: the dataflow specification (includes the set of tasks and their connections), the instance graph, and the clustering specification.

Algorithm 2 Failure of a task-instance $C(i)$

$task_instances \leftarrow \{T(i') | T \in tasks(C) \text{ and } i' \in instances(scope(T)) \text{ such that } i' \rightarrow i \text{ is in the connected instance graph}\}$

$Failure_{cluster}(C, i) \leftarrow \bigcup_{T(i') \in task_instances} Failure_{task}(T, i')$

[0636] **Broker Instance Failure**

[0637] A broker instance failure occurs when a particular broker instance labeled $BROKER(i)$ becomes non-operational. The set of task-instances that will miss incoming messages are the receiver task-instances for any pair of task-instances communicating through the failed broker, plus their (recursively calculated) failures.

[0638] We denote this set as $Failure_{broker}(i)$ as calculated by the following algorithm (Algorithm 3). The set of input specifications to this algorithm are: the dataflow specification (includes the set of tasks and their connections), the instance graph, the clustering specification, and the communication specification.

Algorithm 3 Failure of a broker instance $BROKER(i)$

```

scope ← scope(i)
cluster_pairs ← {(C1, C2) | the clusters C1 and C2 are communicating through scope}
task_pairs ← {(T1, T2) | T1 ∈ tasks(C1), T2 ∈ tasks(C2) and the pair of tasks T1 → T2 in
the datagraph }
task_instances ← {T2(i2) | (T1, T2) ∈ task_pairs, i2 ∈ scope(T2) and i2 → i is in the
connected instance graph }
Failurebroker(i) ←  $\bigcup_{T(i') \in \text{task\_instances}} \text{Failure}_{\text{task}}(T, i')$ 

```

[0639] CPU Failure

[0640] A hardware instance failure occurs when a particular CPU instance labeled $CPU(i)$ become non-operational. As a result, all cluster instances placed on that CPU are failing. In addition, if the CPU is hosting a broker, this broker will fail too. The total set of the task-instances that are failing as result of this CPU failure is the union of all affected task-instances by such local failures.

[0641] We denote this set as $\text{Failure}_{\text{cpu}}(i)$ as calculated by the following algorithm. The set of input specifications to this algorithm are: the dataflow specification (includes the set of tasks and their connections), the instance graph, the clustering specification, the placement specification, and the communication specification.

Algorithm 4 Failure of a CPU instance $CPU(i)$

```

scope ← scope(i)
clusters ← {C | the cluster C is placed on scope }
cluster_instances ← {C(i') | C ∈ clusters and i' ∈ instances(scope(C)) and i' → i is in
the connected instance graph }
Failurecpu(i) ←

```

$$\text{Failure}_{\text{broker}}(i) \bigcup \left(\bigcup_{C(i') \in \text{cluster_instances}} \text{Failure}_{\text{cluster}}(C, i') \right)$$

[0642] Network Connectivity Failure

[0643] A network connectivity failure is a (typically hardware) failure that results in the network connectivity between two CPU instances $CPU(i_1)$ and $CPU(i_2)$ to fail. As a result, all the tasks of all the clusters C placed on the $CPU(i_1)$ that use the broker $BROKER(i_2)$ (if any) for outgoing communications will fail to send messages through this broker.

[0644] We denote this set as $\text{Failure}_{\text{network}}(i_1, i_2)$ as calculated by the following algorithm. The set of input specifications to this algorithm are: the dataflow specification

(includes the set of tasks and their connections), the instance graph, the clustering specification, the placement specification, and the communication specification.

Algorithm 5 Failure of a network link $Network(i_1, i_2)$

```

scope1 ← scope(i1)
scope2 ← scope(i2)
clusters1 ← {C | the cluster C is placed on scope1}
clusters2 ← {C | the cluster C is placed on scope2}
cluster_pairs ← {(C1, C2) | the clusters C1 ∈ clusters1 and C2 are communicating
through scope2}
task_pairs ← {(T1, T2) | T1 ∈ tasks(C1), T2 ∈ tasks(C2) and T1 → T2 in the datagraph}
task_instances ← {T2(j) | (T1, T2) ∈ task_pairs, j ∈ scope(T2) and j → i2 is in the
connected instance graph}
Failurenetwork(i1, i2) ←

```

$$\bigcup_{T(i') \in \text{task_instances}} Failure_{task}(T, i')$$

[0645] Incremental Updates

[0646] Other than building the first version of an application, another (probably more important) source of cost is the application upgrade, versioning and maintenance. This fact is particularly true in the case of IoT systems, where the application is highly distributed, both the software and the hardware components.

[0647] The goal of this section is to explain how our formal model simplifies the problem of up- grade of an application. Our application specification is composed of various components (e.g. the entities, the instances, the operators, the dataflow, the distributed query execution plan, etc). The first goal of this section is to reason about the impact of each potential change to a components in the specification (e.g. deleting a task from the dataflow or moving a certain cluster to the cloud) on the overall system and understanding of the minimal subset of the system (hardware and software) that requires modifications. Ideally, the part of the system that remains untouched should continue to work during upgrades, hence avoiding the situation where an entire application is shut down and restarted upon each update.

[0648] Overview

[0649] Processing of an incremental update consists of several steps. The input to the process is an incremental updates (as defined below). Each update directly changes some data in the specification. These changes introduce consequence sets (an atomic delta on the system) to the rest of the specification, and also consequences to the runtime graph and the system under operation.

[0650] A series of increment updates results in a series of atomic deltas. Applying two deltas one after the other is obviously not a transitive operation; the order in the list of atomic deltas matters.

[0651] In a list of atomic deltas, it is not necessary to apply the consequences set after each atomic delta. A list of atomic deltas, each with its own consequences set, can be "combined" into a single consequences set that implements the combined effect of the entire list of deltas.

[0652] **Deltas and Consequences Sets**

[0653] An *application delta* is an update to be applied to the application configuration or distributed execution plan. Formally, an *application delta* is defined as an ordered list of *atomic deltas*, each restricted to upgrading a specific, identified portion of the application configuration or execution plan.

[0654] *List of atomic deltas.* This section lists all most of the atomic deltas. Such deltas can touch only one clearly identified part of the application, as follows. Additional deltas are straightforward to add.

[0655] *Changes in the application specification.* The following atomic deltas can be applied on the application specification. Each such atomic change is relatively self-descriptive: changes to the application context: changes in the entities, add an entity, remove an unused entity; changes in the operators ontology, change the implementation of an operator, adding a new port to an operator, remove an unused port to an operator. Changes in the dataflow specification: add a task, remove a disconnected task, add a connection, remove a connection, changes of the system parameters of a task (e.g., lineage). Changes of the user parameters: change an operator parameter, change a task parameter, change a task-instance parameter.

[0656] *Changes in the Instances.* Another kind of change happens when the set of instances on which the application is defined changes (e.g. new buildings are added). The set of instances is a tree. It is mathematically well known that every change from a tree to another tree can be expressed as a list of atomic transformations: subtree insertion and subtree addition. Hence, the two set of atomic deltas that we consider here are: add an instance subtree; remove an instance subtree.

[0657] *Changes in the execution plan.* While the application configuration might not change at all, several changes can be made to the distributed execution plan. Atomic changes about the execution plan can be listed as follows: change in the clustering of the dataflow,

merge two clusters into a single cluster, split a cluster in two clusters; change the placement of a cluster, or change the communication hub of two communicating clusters.

[0658] The list of atomic deltas: add-entity(entity, entity-parent), remove-entity(entity), change-implementation(operator, implementation-details), add-port(operator, port), remove-port(operator, port), add-task(task-details), remove-task(task), add-connection(from-task, from-port, to-task, to-port), remove-connection(from-task, from-port, to-task, to-port), change-parameter-operator(operator, parameter, value), change-parameter-task(task, parameter, value), change-parameter-task-instance(task, instance, parameter, value), change-system-parameter-task(task, parameter, value), add-instance(parent-id, new-tree), remove-instance(parent-id, child-id), split-clusters(C1, C2, C3), merge-clusters(C1, C2, C3), change-placement(C, entity), change-communication(C1, C2, entity)

[0659] *Defining the atomic deltas.* For each atomic delta in the list above, we will specify: the primary yaml specification that needs to be modified, the formal definition of the delta, a yaml formatting of the delta.

[0660] Modifying a single concept in the application specification might trigger a cascade of other modifications in other places in order to keep the specification correct, and executable. Sometimes, removing an item (e.g. task) from the application specification needs removing it from other places where this item is being used or referenced (e.g. clustering and placement, eventually).

[0661] In addition, the deltas that change the application specification obviously have consequences over the query execution plan. For each such change we will provide a default¹ execution plan for the modified application. This feature means that more specifications may potentially be modified as result.

[0662] So, in addition to the primary yaml specification that we describe above, we will also specify two other lists: the list of other yaml specifications that need to be modified to avoid referring to missing concepts, and the list of query execution plans yaml specifications that need to be modified in order to keep the execution plan correct and executable.

[0663] Moreover, it is possible that some concepts (brokers, CPUs) might remain unused after the modification is applied. After each such atomic update we will implicitly apply a garbage collection method, which will have its own list of modifications to the yaml cpus.yaml. Such cases are called out explicitly.

[0664] Finally, some concepts are being used in various places in the application specification. For example, an entity can be used as a scope of a task, as a placement of a cluster, or as a communication hub for two clusters. In case the removal of such entity is

requested, we will not apply such updates automatically. Instead, we will simply suggest the list the atomic updates that need to be applied manually by the user *before* the removal, in order to make this update possible.

[0665] *Defining the consequences set.* Given a particular atomic delta, we calculate the minimum set of changes that need to be applied to the software and hardware infrastructure, that we name the *consequences set* of that given delta. We note that the first kind of updates apply uniformly to the hardware or software concepts (CPU, broker or cluster) associated with all instances of an entity. In other words, if the software or hardware associated with an instance is updated in a particular way, then all the instances of the same entity are updated, in the same way.

[0666] The *consequences set* of a delta is formally defined as seven sets, as described below.

[0667] The sets describe changes in: the *hardware infrastructure*, via the two sets:
newCPU (entity) : identifies the set of entities whose instances have a new CPU added;
deleteCPU (entity) : identifies the set of entities whose instances have CPUs that need to be deleted from the hardware network. The *software components* of the application. Given that at runtime the application is composed of (a) containers, each corresponding to a cluster instance and identified by a pair *Cluster (instance)*, for each *instance* in the scope of the *Cluster* , and (b) brokers, each identified as *Broker(instance)*. The modifications to the software infrastructure can be described by the following five sets: *newBroker (entity)* : identifies the set of entities whose instances need to received a new Broker, *deleteBroker (entity)* : identifies the set of entities whose instances need to delete their existing Broker, *newCluster (cluster)* : contains a set of clusters and identifies the new clusters whose instances need to be added to the software infrastructure, *deleteCluster (cluster)* : identifies the clusters whose instances need to be deleted from the software infrastructure, *updatedCluster (cluster, kind)* : identifies the clusters whose instances need to be updated in the software infrastructure, together with the *kind* of the upgrade. The *kind* field is one of the constants: "parameters" if only the parameters of some tasks in the container need updates, "data" if the data of the cluster instance is updated, but not the code, and "reload" if more components, most likely including the code needs to be upgraded.

[0668] We distinguish between those cases because they may allow for simpler upgrades: *parameters* change may allow for parameters changes while the cluster is still running. Hence, this can be done by normal message processing, and does not require more

sophisticated upgrade mechanisms. *Data* implies that some data portions of the cluster content change, but not the code. This can be either (a) the data that describes the instance subtree on which this cluster applies or (b) the dataflow subgraph on which the cluster operates. Such case requires the upload of a potentially smaller quantity of data. However, this does require shutting off and the restart of the cluster instances. *Reload* in case that the entire cluster content needs to be reloaded. This obviously implies stopping and restarting the newly reloaded cluster instance.

[0669] Prerequisites

[0670] The algorithms that follow use definitions and notions that have been defined in the previous document [4]. We repeat these definitions here so that this document is self-contained. The computation uses the following sets and functions from the original and/or new specification: *root* = the root of the entity hierarchy, *tasks* = the set of tasks of the application, *clusters* = the set of clusters in the execution plan, *cpus* = the set of entities that hold CPUs, *brokers* = the set of entities that hold brokers, *placement* (*C*) = the placement of a cluster *C*, *communication*(*C*₁, *C*₂) = the communication hub of a pair of clusters *C*₁ and *C*₂, *tasks* (*C*) = the set of tasks in the cluster *C*, *cluster* (*T*) = the cluster of the task *T*, *entity* (*i*) = the entity the instance *i* belongs to, *scope* (*T*) = the scope of a task *T*, *scope* (*C*) = the scope² of cluster *C*, *instances* (*e*) = the instances of the entity *e*, *instances* (*C*) = *instances* (*scope* (*C*)) all the instances of the cluster *C*, *ascendants* (*i*) = the instances *i*' that are ascendants of *i* in the instance graph (including *i*), *descendants* (*i*) = the instances *i*' that are descendants of *i* in the instance graph (including *i*), *connected* (*i*) = *ascendants* (*i*) $\cup$ *descendants* (*i*), *ascendants* (*e*) = the entities *e*' that are ascendants of *i* in the instance graph (including *i*), *descendants* (*e*) = the entities *e*' that are descendants of *i* in the instance graph (including *i*), *connected* (*e*) = *ascendants* (*e*) $\cup$ *descendants* (*e*), *instance_placement* (*C*, *i*) = *instances* (*placement* (*C*)) $\cap$ *ascendants* (*i*), *connected* (*T*₁, *T*₂) = true IFF the tasks *T*₁ and *T*₂ are connected in the dataflow graph

[0671] Consequences Set Calculations for Query Execution Plan Changes

[0672] *Merge two clusters. Delta.* The part of the specification that is being modified is: clustering.yaml

[0673] Formally, this atomic delta is defined as a triple of clusters (*C*₁, *C*₂, *C*₃). After the modification the cluster *C*₁ and *C*₂ are deleted, and a new cluster *C*₃ is added, containing both the tasks of *C*₁ and *C*₂.

[0674] An example of such a delta is merging the clusters *cluster_one* and *cluster_two* into a new cluster *cluster_three*. This delta can be expressed in a yaml format as follows.

```
- merge-clusters:
  - clusters: [cluster_one, cluster_two]
  - result-cluster: cluster_three
```

[0675] Fig. 56 shows yaml 5600 describing merging the cluster one and cluster two in clustering.yaml. The resulting (merged) cluster has a default³ placement on *root*. Also, all (old and new) communications of the newly merged cluster *C3* will be by default using the Broker of the *root* entity.

[0676] As the result of this update, the following other yamls needs to be automatically updated: placement.yaml, communication.yaml as follows: in placement.yaml, placement(*C*₁, *_*) is deleted, placement(*C*₂, *_*) is deleted, placement(*C*₂, "root") is added, in communication.yaml. For every cluster *C* (*C* ≠ *C*₁ and *C* ≠ *C*₂) replace communication(*C*, *C*₁, *_*) with communication(*C*, *C*₃, "root") and communication(*C*, *C*₂, *_*) with communication(*C*, *C*₃, "root"), delete communication(*C*₁, *C*₂, *_*)

[0677] *Consequences set*. The consequences sets calculates all the hardware and software modifications that need to be applied to a running application in order to implement that delta.

[0678] First we need to compute some temporary helper variables.

other_clusters = *clusters* \ {*C*₁, *C*₂}

other_connected_clusters = {*C* | *C* ∈ *other_clusters* such that ∃ *T*₁ ∈ *tasks* (*C*), *T*₂ ∈ *tasks* (*C*₃)

such that *connected* (*T*₁, *T*₂)}

[0679] Now we can calculate the consequences set.

deleteCluster = {*C*₁, *C*₂}

newCluster = {*C*₃}

updateCluster = {(*C*, "parameters") | *C* ∈ *other_connected_clusters*}

[0680] After this update, some brokers and/or CPUs may be left unused. Hence, we apply the garbage collecting method described bellow.

[0681] *Garbage collecting unused clusters, brokers and CPUs*. This method is automatically triggered after each update that can let any of the brokers or CPU unused.

Automatically, those need to be registered and deleted from the infrastructure. Hence, potentially, this such may trigger additional modifications in `cpu.yaml`.

[0682] First we need to compute some temporary helper variables.

$$\begin{aligned} used_CPU\ s &= \{placement^{after}(C) | C \in clusters^{after}\} \\ &\cup \{communication^{after}(C_1, C_2) | C_1, C_2 \in clusters^{after} \text{ and } C_1 \text{ and } C_2 \text{ are connected}\} \\ all_Brokers^{before} &= \{communication^{before}(C_1, C_2) | C_1, C_2 \in clusters^{before}\} \\ all_Brokers^{after} &= \{communication^{after}(C_1, C_2) | C_1, C_2 \in clusters^{after}\} \end{aligned}$$

[0683] The garbage collection augments the consequences set of the atomic updates it is applied after as follows:

$$\begin{aligned} deleteBroker &= deleteBroker \cup ((all_Brokers^{before} \setminus all_Brokers^{after}) \setminus \{root\}) \\ deletedCPU &= deleteCPU \cup ((cpus^{before} \setminus used_CPU\ s) \setminus \{root\}) \end{aligned}$$

[0684] The garbage collection method will be triggered automatically. We will mark explicitly the updates that need to invoke it.

[0685] *Split a cluster in two clusters. Delta.* The part of the specification that is being modified is: `clustering.yaml`.

[0686] Formally, the delta is defined as a triple of clusters ($C_1, C_2 = tasks(C_2), C_3 = tasks(C_3)$). After the modification the cluster C_1 is deleted and new clusters C_1 and C_2 are created; the tasks of the cluster C_1 are split between the clusters C_2 and C_3 .

[0687] An example of such a delta is splitting of the cluster `cluster_two` into `cluster_three` and `cluster_four`. This delta can be expressed in a `yaml` format as follows.

```
- split-cluster:
  - input-cluster: cluster_two
  - output
    - cluster:
        name: cluster_three
        tasks: [crop, recognize]
    - cluster:
        name: cluster_four
        tasks: [enrich, monitor]
```

[0688] Fig. 57 shows `yaml 5700` describing splitting `cluster_two` in `clustering.yaml`. The resulting clusters have a default placement on `root`. Also, all (old and new) communications of the newly created clusters C_2 and C_3 will be by default using the Broker of the `root` entity.

[0689] As the result of this update, the following other `yamls` needs to be automatically updated: `placement.yaml`, `communication.yaml` as follows: in `placement.yaml`, `placement(C1,`

$_)$ is deleted, $\text{placement}(C_2, \text{"root"})$ is added, $\text{placement}(C_2, \text{"root"})$ is added, in $\text{communication.yaml}$, delete $\text{communication}(C_1, _, _)$, for every cluster C that after the update communicates with C_2 add $\text{communication}(C, C_1, \text{"root"})$, for every cluster C that after the update communicates with C_3 add $\text{communication}(C, C_3, \text{"root"})$.

[0690] *Consequences set.* The consequences sets calculates all the hardware and software modifications that need to be applied to a running application in order to implement that delta.

[0691] First we need to compute some temporary helper variables.

$\text{other_clusters} = \text{clusters} \setminus \{C_1\}$

$\text{other_connected_clusters} = \{C \mid C \in \text{other_clusters} \text{ such that } \exists T_1 \in \text{tasks}(C), T_2 \in \text{tasks}^{\text{before}}(C_1)$

such that $\text{connected}^{\text{before}}(T_1, T_2)\}$

[0692] Now we can calculate the consequences set.

$\text{deleteCluster} = \{C_1\}$

$\text{newCluster} = \{C_2, C_3\}$

$\text{updateCluster} = \{(C, \text{"parameters"}) \mid C \in \text{other_connected_clusters}\}$

[0693] Garbage collection (Section 9.4.1) needs to be invoked at the end.

[0694] *Change the placement of a cluster. Delta.* The part of the specification is that is being modified is: placement.yaml .

[0695] Formally, the delta is defined as a pair (C, e) . After the modification each cluster instances (C, i) is moved to the CPU instance $\text{CPU}(i')$ such that i' is an instance of the entity e that is an ascendant of i .

[0696] An example of such a delta is moving the cluster.

```
- move-cluster:
  name: cluster_two
  placement: root
```

[0697] Fig. 58 shows an example placement update 5800. Replacing a cluster might invalidate some cluster communication decisions. Hence, by default all the communications of the re-placed cluster are automatically moved to the *root*.

[0698] As the result of this update, the following other yaml needs to be automatically updated: $\text{communication.yaml}$ as follows: for every cluster $C_1 \neq C$ replace $\text{communication}(C, C_1, _)$ with $\text{communication}(C, C_1, \text{"root"})$, and for every cluster $C_1 \neq C$ replace $\text{communication}(C_1, C, _)$ with $\text{communication}(C_1, C, \text{"root"})$.

[0699] *Consequences set.* The consequences sets calculates all the hardware and software modifications that need to be applied to a running application in order to implement that delta.

[0700] First we need to compute some temporary helper variables.

$other_clusters = clusters \setminus \{C\}$

$other_connected_clusters = \{C_1 | C_1 \in other_clusters \text{ such that } \exists T_1 \in tasks(C_1), T_2 \in tasks(C)$

such that $connected(T_1, T_2)\}$

[0701] The consequences set of this particular update is the following.

$deleteCluster = \{C_1\}$

$newCluster = \{C_2, C_3\}$

$updateCluster = \{(C, "parameters") | C \in other_connected_clusters\}$

[0702] Garbage collection is invoked at the end.

[0703] *Change the communication hub of two communicating clusters. Delta.* The part of the specification that is being modified is: communication.yaml.

[0704] Formally, the delta is defined as a triple (C_1, C_2, e) . After the modification all communications between any task-instance (T_1, i_3) belonging to cluster (C_1, i_1) and any other task-instance (T_2, i_4) belonging to cluster (C_2, i_2) will use the broker instance *Broker* (i') such that i' is an instance of the entity e that is an ascendant of both i_1 and i_2 . Of course, such update is only legal if e is an ascendant of both *placement* (C_1) and *placement* (C_2). An example of such a delta is moving the cluster.

```
- move-communication:
  clusters:
    first: cluster_one
    second: cluster_second
  communication: root
```

[0705] Fig. 59 shows an example a communication update 5900. *Consequences set.* The consequences sets calculates all the hardware and software modifications that need to be applied to a running application in order to implement that delta.

[0706] The consequences set of this particular update is the following.

$updateCluster\ Instance = \{(C_1, "parameters")\} \cup \{(C_2, "parameters")\}$

[0707] Garbage collection is invoked at the end.

[0708] Consequences Set Calculations for Dataflow Updates

[0709] *Add a connection. Delta.* The part of the specification that is being modified is: dataflow.yaml.

[0710] Formally, the delta is defined as a tuple $(from_task, from_port, to_task, to_port)$. The update adds a new data communication link between the two tasks. This delta can be expressed in a yaml format as follows.

- add-connection:
 - from: {task: heartbeat, output_port: heartbeat}
 - to: {task: camera, input_port: heartbeat}

[0711] Fig. 60 shows example yaml 6000 describing adding a new task connection in dataflow.yaml. The new task communication link may add a new cluster communication, in case the two tasks belong to different clusters. If this is the case, the communication hub between the two clusters is "root".

[0712] As the result of this update, the following other yamls needs to be automatically updated: communication.yaml as follows: if $cluster(from_task) \neq cluster(to_task)$ and $connected^{before}(cluster(from_task), cluster(to_task))$ is false then add $communication(cluster(from_task), cluster(to_task), "root")$

[0713] *Consequences set.* The consequences sets calculates all the hardware and software modifications that need to be applied to a running application in order to implement that delta. First we need to compute some temporary helper variables.

$$other_clusters^{before} = \{cluster(T) | T \in tasks \text{ and } connected^{before}(T, to_task)\}$$

$$other_clusters^{after} = \{cluster(T) | T \in tasks \text{ and } connected^{after}(T, to_task)\}$$

$$from_cluster = other_clusters^{after} \setminus other_clusters^{before}$$

[0714] Now we can calculate the consequences set⁴.

$$updateCluster = \{C, "data" | C \in from_cluster \cup cluster(to_task)\}$$

[0715] *Remove a connection. Delta.* The part of the specification that is being modified is: dataflow.yaml.

[0716] Formally, the delta is defined as a tuple $(from_task, from_port, to_task, to_port)$. The update removes a data communication link between two tasks. This delta can be expressed in a yaml format as follows.

- remove-connection:
 - from: {task: heartbeat, output_port: heartbeat}
 - to: {task: camera, input_port: heartbeat}

[0717] Fig. 61 shows example yaml 6100 describing removing a task connection in dataflow.yaml. The removed task communication link is potentially removing a cluster

communication. Hence, the following other yamls needs to be automatically updated:
communication.yaml as follows: in communication.yaml remove *connection(cluster (from_task)), cluster (totask), _)* if *cluster (from_task) ≠ cluster (to_task)* and for every task $T_1 \in \text{tasks (from_task)}$ and $T_2 \in \text{tasks (to_task)}$ *connected^{after} (T₁, T₂)* is *false*

[0718] *Consequences set.* The consequences sets calculates all the hardware and software modifications that need to be applied to a running application in order to implement that delta.

[0719] First we need to compute some temporary helper variables. First we need to compute some temporary helper variables.

$\text{other_clusters}^{\text{before}} = \{\text{cluster (T)} | T \in \text{tasks and connected}^{\text{before}} (T, \text{to_task})\}$

$\text{other_clusters}^{\text{after}} = \{\text{cluster (T)} | T \in \text{tasks and connected}^{\text{after}} (T, \text{to_task})\}$

$\text{from_cluster} = \text{other_clusters}^{\text{after}} \setminus \text{other_clusters}^{\text{before}}$

[0720] Now we can calculate the consequences set⁵.

$\text{updateCluster} = \{C, \text{"data"}\} | C \in \text{from_cluster} \cup \text{cluster (to_task)}\}$

[0721] Garbage collection (Section 9.4.1) needs to be invoked at the end.

[0722] *Add a task. Delta.* The part of the specification is that is being modified is: dataflow.yaml

[0723] Formally, the delta is defined as a task definition *T* that includes task name, the task operator and its scope. The update adds a new task to the dataflow; the new task is not yet linked to any other task. This delta can be expressed in a yaml format as follows.

```
- add-tasks: {
  name: camera,
  operator: CameraSensorPiCamera, scope: door }
```

[0724] Fig. 62 shows example yaml 6200 describing adding a new task in dataflow.yaml.

From the execution point of view, the new task will create a cluster of its own called *T _cluster*, placed at the *root*. As the result of this update, the following other yamls needs to be automatically updated: placement.yaml, communication yaml as follows: clustering yaml - *cluster (T , T _cluster)* is added, placement.yaml - *placement (T _cluster, "root ")* is added.

[0725] *Consequences set.* The consequences sets calculates all the hardware and software modifications that need to be applied to a running application in order to implement that delta. Now we can calculate the consequences set.

$\text{newCluster} = \{T_cluster\}$

[0726] *Remove a disconnected task. Delta.* The part of the specification is that is being modified is: dataflow.yaml.

[0727] Formally, the delta is a single task name T . The update can only be performed if there are no connections to or from the task T 's ports. If this condition is not true, then suggest the list of updates: for all connection from or to the task T , remove-connection(). This delta can be expressed in a yaml format as follows.

```
- remove-task:
  task: heartbeat
```

[0728] Fig. 63 shows example yaml 6300 describing removing a task in dataflow.yaml. The removed task is also potentially removing a cluster. As the result of this update, the following other yamls needs to be automatically updated: clustering.yaml and communication.yaml via if $tasks_{after}(cluster(T)) = \emptyset$ then: remove $cluster(T)$, remove placement($cluster(T)$)

[0729] *Consequences set.* The consequences set is as follows:

$updateCluster = \{(cluster(T), "reload") \text{ if } tasks_{after}(cluster(T)) \neq \emptyset\}$

$deleteCluster = \{(cluster(T) \text{ if } tasks_{after}(cluster(T)) = \emptyset\}$

[0730] Garbage collection (Section 9.4.1) needs to be invoked at the end.

[0731] *Change an operator parameter. Delta.* The part of the specification is that is being modified is: parameters.yaml

[0732] Formally, the delta is a triple $(O, parameter, value)$ where O is an operator name. This delta can be expressed in a yaml format as follows.

```
- change-operator-parameter:
  operator-name: CameraPiSensor
  parameters
  - parameter
    name: width
    value: 640
```

[0733] Fig. 64 shows example yaml 6400 describing changing an operator parameter value in parameters.yaml. We need to do the following calculations:

$touched_tasks = \{T | T \in tasks \text{ such that } operator(T) = O\}$

$touched_clusters = \{cluster(T) | T \in touched_tasks\}$

[0734] *Consequences set.* The consequences set is as follows:

$updateCluster = \{(C, "parameters") | C \in touched_clusters\}$

[0735] *Change a task parameter. Delta.* The part of the specification is that is being modified is: parameters.yaml

[0736] Formally, the delta is a triple $(T, \text{parameter}, \text{value})$ where T is a task name. This delta can be expressed in a yaml format as follows.

```
- change-task-parameter:
  task-name: camera
  parameters
  - parameter
    name: width
    value: 640
```

[0737] Fig. 65 shows example yaml 6500 describing changing a task parameter value in parameters.yaml. The consequences set is as follows:

updateCluster = $\{(cluster(T), "parameters")\}$

[0738] *Change the system properties of a task. Delta.* The part of the specification is that is being modified is: tasks.yaml

[0739] Formally, the delta is a triple $(T, \text{parameter}, \text{value})$ where T is a task name and *parameter* is a system parameter (e.g. lineage). This delta can be expressed in a yaml format as follows.

```
- change-task-system-parameter:
  - task: camera
  - system-parameter:
    provenance: true
```

[0740] Fig. 66 shows example yaml 6600 describing changing a task system parameter in tasks.yaml.

[0741] Consequences set. The consequences set is as follows:

updateCluster = $\{(cluster(T), "reload")\}$

[0742] **Consequences Set Calculation for Context Updates**

[0743] *Add an entity. Delta.* The part of the specification is that is being modified is: entities.yaml

[0744] Formally, the delta is defined as a pair of entities (e_1, e_2) where e_1 is the new entity to be added and e_2 is the parent to which e_1 is to be added as a child. This delta can be expressed in a yaml format as follows.

```
add-entity:
  name: building
  parent: room
```

[0745] Fig. 67 shows an example for adding a new entity update 6700. *Consequences set.* The consequences set is empty.

[0746] *Remove an unused entity. Delta.* The part of the specification is that is being modified is: entities.yaml

[0747] Formally, the delta is defined as an entity e to be deleted. The update can only be performed if the given entity is not used anywhere: as a parent for other entities, as a scope for a task, as a placement for a cluster, or as a communication hub. If this condition is not true, then suggest the list of following updates to be performed prior to this update.

```
{remove__entity(e1)|e1 ∈ children(e)}
{change__communication(C1, C2, "root ")|communication(C1, C2, e)}
{change__placement(C, "root ")|placement(C, e)}
{remove__task(T)|scope(T) = e }
```

[0748] This delta can be expressed in a yaml format as follows.

remove-entity:

```
name: building
```

[0749] Fig. 68 shows an example for removing 6800 an entity. *Consequences set.* The consequences set is empty.

[0750] *Change the implementation of an operator. Delta.* The part of the specification is that is being modified is: operators.yaml

[0751] Formally, the delta is a tuple $(O, parameter, value)$ where O is an operator being modified, followed by the rest of details necessary (e.g. class, includes, etc).

```
- change-operator-implementation:
  operator-name: CameraPiSensor
  implementation:
    language: python
    module: demo.tasks.camera_sensor_pi_camera
  pip:
    - "picamera==1.13"
    - "Pillow==8.3.2"
```

[0752] Fig. 69 shows example yaml 6900 describing changing an operator implementation in operators.yaml. We need to do the following calculations:

$touched_tasks = \{T | T \in tasks \text{ such that } operator(T) = O\}$

$touched_clusters = \{cluster(T) | T \in touched_tasks\}$

[0753] *Consequences set.* The consequences set is as follows:

$updateCluster = \{(C, "reload") | C \in touched_clusters\}$

[0754] *Add a port to an operator. Delta.* The part of the specification is that is being modified is: operators.yaml

[0755] Formally, the delta is defined a pair (*operator*, *port*, *kind*) specifying the new port to be added to the specified operator, together with it's *kind*, which can be one of the following: {*send*, *process*, *compute*, *request* }.

[0756] This delta can be expressed in a yaml format as follows.

add-operator:

```
operator: building
port: new_port
kind: "process"
```

[0757] Fig. 70 shows an example for adding a new port to an operator 7000. *Consequences set*. The consequences set is empty.

[0758] *Remove an unused port of an operator. Delta*. The part of the specification is that is being modified is: operators.yaml

[0759] Formally, the delta is defined a pair (*operator*, *port*) specifying the port to be removed from the specified operator. The update can only be performed if there are no connections to or from the task given port. If this condition is not true, then suggest the list of updates: for all connection from or to the task T on the given port, remove-connection(). This delta can be expressed in a yaml format as follows.

remove-operator:

```
operator: building
port: deleted_port
```

[0760] Fig. 71 shows an example 7100 for removing a port from an operator.

Consequences set. The consequences set is empty.

[0761] Merging Lists of Atomic Updates

[0762] Given an update defined as a list of atomic updates, we will perform the following algorithm in order to decide the final version of the consequence set.

Algorithm 6 Collapsing a list of atomic updates

```

collapse( $consequence\_set^{before}$ ,  $specification^{before}$ , list of atomic updates  $L$ ):
  if  $L$  is empty then
    return ( $specification^{before}$ ,  $consequence\_set^{before}$ )
  else
     $update \leftarrow head(L)$ 
     $specification^{after} = apply(specification^{before}, update)$ 
     $consequence\_set^{after} \leftarrow consequence\_set(specification^{before}, specification^{after},$ 
     $update)$ 
     $merge\_consequences \leftarrow merge(consequence\_set^{before}, consequence\_set^{after})$ 
    return collapse( $merge\_consequences$ ,  $specification^{after}$ , tail( $L$ ))
  end if

```

[0763] The recursive loop starts with algorithm 7.

Algorithm 7 Collapsing a list of atomic updates: trigger the recursive loop

```

collapseStart( $specification$ , list of atomic updates  $L$ ):
  return collapse(empty_consequence_set,  $specification$ ,  $L$ )

```

[0764] *Merging two consequence sets.* Given two consequences sets, called $consequences^1$ and $consequences^2$, the previous procedure invokes merging them into a combined consequence set. This is done by the algorithm bellow.

Algorithm 8 Merging two consequences sets, Part 1

```

merge(consequences1, consequences2):
    deleteCPUfinal ← ∅
    newCPUfinal ← ∅
    deleteBrokerfinal ← ∅
    newBrokerfinal ← ∅
    deleteClusterfinal ← ∅
    newClusterfinal ← ∅
    updateClusterfinal ← ∅
    for all e in deleteBroker1 do
        if e in deleteBroker2 then
            error
        else if e in newBroker2 then
            nothing
        else
            deleteBrokerfinal += {e}
        end if
    end for
    for all e in newBroker1 do
        if e in deleteBroker2 then
            nothing
        else if e in newBroker2 then
            error
        else
            newBrokerfinal += {e}
        end if
    end for
    for all e in deleteCPU1 do
        if e in deleteCPU2 then
            error
        else if e in newCPU2 then
            nothing
        else
            deleteCPUfinal += {e}
        end if
    end for
    for all e in newCPU1 do
        if e in deleteCPU2 then
            nothing
        else if e in newCPU2 then
            error else
            newCPUfinal += {e}
        end if
    end for

```

57

Algorithm 9 Merging two consequences sets. Part 2

```
for all e in newCluster2 do
  if e in newCluster2 then
    error
  else if e in deleteCluster2 then
    nothing
  else
    newClusterfinal += {e}
  end if
end for
for all e in deleteCluster1 do
  if e in deleteCluster2 then
    error
  else if e in newCluster2 then
    nothing
  else if e in updateCluster2 then
    error
  else
    deleteClusterfinal += {e}
  end if
end for
for all e in updateCluster1 do
  if e in newCluster2 then
    error
  else if e in deleteCluster2 then
    deleteClusterfinal += {e}
  else if e in updateCluster2 then
    updateClusterfinal += {(e, kindfinal)}
  else
    updateClusterfinal += {(e, kind1)}
  end if
end for
```

[0765] In the case when an entity is updated twice, the kind of update needs to be obtained by "merging" the two individual kinds. The merging table is given below.

c	data	parameters	reload
data	data	data	reload
parameters	data	parameters	reload
reload	reload	reload	reload

[0766] Instance Sensitive Incremental Updates

[0767] *Defining the instance sensitive consequences set.* Given a particular atomic delta, we calculate the minimum set of changes that need to be applied on the software and hardware infrastructure, that we name the consequences set of that given delta.

[0768] The *consequences set* of a delta is formally defined as seven sets, as described below. The sets describe changes in the *hardware infrastructure*, via the two sets: *newCPU Instance (instance)* : contains a set of instance identifiers and identifies the new CPUs that need to be added to the hardware network as result of the delta, *deleteCPU Instance (instance)* : identifies the CPUs that need to be deleted from the hardware network as result of the delta.

[0769] The *software components* of the application. Given that at runtime, the application is composed of (a) containers, each corresponding to a cluster instance and identified by a pair *Cluster (instance)*, and (b) brokers, each identified as *Broker(instance)* the modifications to the software infrastructure can be described by the following five sets: *newBroker Instance (instance)* : identifies the set of CPU instances that need to receive a new Broker, *deleteBroker Instance (instance)* : identifies the set of CPU instances that need to delete their existing Broker, *newCluster Instance (cluster, cluster_instance, CPU_instance)* : contains a set of triples (*cluster, instance, CPU_instance*) and identifies the new cluster instances *cluster (instance)* that need to be added to the software infrastructure, and that this new cluster instance needs to be installed on the CPU identified by *CPU_instance*, *deleteCluster Instance (cluster, instance, CPU_instance)* : identifies the cluster instances *cluster (instance)* that need to be deleted from the software infrastructure, and that they have been previously located on the CPU identified by *CPU_instance*, *updateCluster Instance (cluster, instance, old_CPU_instance, new_CPU_instance, content_change)* : identifies the cluster instances, *cluster (instance)* that need to be upgraded within the software infrastructure. The update change follows the same rules as in the case of uniform updates.

[0770] Note that the algorithms use those definitions from both the application specification *before* the application of the update, and *after* the modification of the application specification. We will mark the two variants using the *before* or *after* markers to make the difference between the two variants. In case some definitions do not change in the two versions; in those cases we simply omit using the *before* or *after* markers. Also note that this refers to the original user application specifications, before any automatic preprocessing

of any kind (e.g. adding Adaptors in order to implement the correct distributed execution) has been applied to it.

[0771] Also, the discussion in the remainder of this paper is based on a main assumption: that the entity hierarchy has a unique root called *root*, and that the entity root is *always* hosting both a CPU and a Broker (that would be the equivalent of *cloud* computing, which is always available). Hence we have the following statements always true: $root \in cpus$, $root \in brokers$

[0772] Consequences Set Calculation for Instance Updates

[0773] *Add an instance subtree. Delta.* The part of the specification is that is being modified is: `instances.yaml`

[0774] Formally, the delta is defined as a pair of instance identifiers (id_1, id_2) , where a new subtree rooted as instance identified with id_1 is added as a child to the instance identified with id_2 (id_2 being an instance from the instance tree before the modification).

[0775] An example of such a delta is adding a new room with one door to the building #1. This delta can be expressed in a yaml format as follows.

- added-subtree:
 - relationship:
 - name: has-a
 - from: building/1/
 - to: building/1/room/3/
- new-subtree:
 - instances:
 - instance: {id: building/1/room/3/, entity: room}
 - instance: {id: building/1/room/3/door/1/, entity: door}
 - relationships:
 - relationship:
 - name: has-a
 - from: building/1/room/3/
 - to: building/1/room/3/door/1/

[0776] Fig. 72 shows example yaml 7200 describing adding a new subtree to the `instances.yaml`. No other yaml specifications need to change as result of this atomic change. The new task-instances that are implicitly added to the runtime graph by this instances insert might receive additional task-instance parameters, but this can be expressed as an additional, separate atomic delta.

[0777] *Consequences set.* The consequences sets calculates all the hardware and software modifications that need to be applied to a running application in order to implement that delta.

[0778] First we need to compute some temporary helper variables.

$new_instances = descendants(i_1)$

$touched_entities = \{entity(i) | i \in new_instances\}$

$touched_clusters = \{C | C \in clusters \text{ and } \exists \text{ a task } T \in tasks(C) \text{ such that } scope(T) \in touched_entities\}$

$included_clusters = \{C | C \in touched_clusters \text{ such that } scope(C) \in descendants(entity(i_1))\}$

$modified_clusters = touched_clusters \setminus included_clusters$

[0779] The set of new CPU instances that need to be added to the infrastructure are calculated as follows. The set of CPU instances that need to be deleted is obviously empty.

$newCPU = \{i | i \in new_instances \text{ such that } entity(i) \in cpus\}$

$deletedCPU = \emptyset$

[0780] The set of new broker instances that need to be added to the software infrastructure are calculated as follows. The set of brokers instances that need to be deleted is empty.

$newBrokers = \{i | i \in new_instances \text{ such that } entity(i) \in brokers\}$

$deletedBrokers = \emptyset$

The set of new cluster instances that need to be added to the software infrastructure (together with the CPU instances that need to host them) and the set of clusters instances that need to be modified are calculated as follows. Naturally, no cluster instances need to be deleted.

$newClusterInstance = \{(C, i_3, i_4) | C \in included_clusters, i_3 \in instances^{after}(C) \cap new_instances \text{ and } i_4 = instance_placement^{after}(C, i_3)\}$

$updateClusterInstance = \{(C, i_3, i_4, null, "data") | C \in modified_clusters,$

$i_3 \in instances(C) \cap ascendants(i_1) \text{ and}$

$i_4 = instance_placement^{before}(C, i_3)\}$

$deletedClusterInstance = \emptyset$

[0781] The calculation of the set *updateClusterInstance* means that all "modified" cluster instances need to be stopped and re-launched. The modification concerns only the graph instance data component of the cluster instance, and not the code of the cluster; hence marked as a constant "data" field in the tuple. The CPU instance location of those cluster instances remains unchanged (which is marked by a *null* value for the new CPU instance).

[0782] *Post-updates.* The set of post-updates is empty.

[0783] *Delete an instance subtree. Delta.* The part of the specification is that is being modified is: instances.yaml

[0784] Formally, the delta is defined as a pair of instance identifiers (id_1, id_2) , where the subtree rooted as instance identified with id_1 is deleted as a child of the instance identified with

[0785] id_2 (id_1 and id_2 being instances from the instance tree before the modification).

[0786] An example of such a delta is deleting the room #3 from the building #1. This delta can be expressed in a yaml format as follows.

```
- deleted-subtree:
  - relationship:
      name: has-a
      from: building/1/
      to: building/1/room/3/
```

[0787] Fig. 73 shows example yaml 7300 describing deleting a subtree from the instances.yaml. No other yaml specifications need to change as result of this atomic change.

[0788] *Consequences set.* The consequences sets calculates all the hardware and software modifications that need to be applied to a running application in order to implement that delta.

[0789] First we need to compute some temporary helper variables.

$deleted_instances = descendants(i_1)$

$touched_entities = \{entity(i) | i \in deleted_instances\}$

$touched_clusters = \{C | C \in clusters \text{ and } \exists T \in tasks(C) \text{ such that } scope(T) \in touched_entities\}$

$included_clusters = \{C | C \in touched_clusters \text{ such that } scope(C) \in descendants(entity(i_1))\}$

$modified_clusters = touched_clusters \setminus included_clusters$

[0790] The set of CPU instances that need to be deleted from the infrastructure are calculated as follows. The set of new CPU instances that need to be added is obviously empty.

$newCPU = \emptyset$

$deletedCPU = \{i | i \in deleted_instances \text{ such that } entity(i) \in cpus\}$

[0791] The set of broker instances that need to be shut down in the software infrastructure are calculated as follows. The set of brokers instances that need to be added is obviously empty.

$newBrokers = \emptyset$

$deletedBrokers = \{i \mid i \in new_instances \text{ such that } entity(i) \in brokers\}$

[0792] The set of cluster instances that need to be deleted from the software infrastructure (together with the CPU instances that need to host them) and the set of clusters instances that need to be modified are calculated as follows. Naturally, no cluster instances need to be added.

$newClusterInstance = \emptyset$

$updateClusterInstance = \{(C, i_3, i_4, null, "data") \mid C \in modified_clusters,$

$i_3 \in instances(C) \cap ascendants(i_1) \text{ and}$

$i_4 = instances_placement^{before}(C, i_3)\}$

$deletedClusterInst = \{(C, i_3, i_4) \mid C \in included_clusters, i_3 \in instances^{before}(C) \cap$

$deleted_instances \text{ and } i_4 = instances_placement^{before}(C, i_3)\}$

[0793] The set *updateClusterInstance* marks all "modified" cluster instances need to be stopped and re-launched. The modification concerns only the graph instance data component of the cluster instance, and not the code of the cluster; hence marked as a constant "data" field in the tuple. The CPU instance location of those cluster instances remains unchanged (which is marked by a *null* value for the new CPU instance).

[0794] *Post-updates.* The set of post-updates is empty.

[0795] *Change a task-instance parameter. Delta.* The part of the specification is that is being modified is: parameters.yaml

[0796] Formally, the delta is a tuple $(T, i, parameter, value)$ where T is a task name and i is an instance identifier. This delta can be expressed in a yaml format as follows.

```
- change-task-instance-parameter:
  task-name: camera
  instance_id: building/1/room/1/door/1/
  parameters
  - parameter
    name: width
    value: 640
```

[0797] Fig. 74 shows example yaml 7400 describing changing a task-instance parameter value in parameters.yaml. The consequences set is as follows: *updateCluster* = {(cluster (T), "parameter ")}

[0798] *Post-updates*. The set of post updates is empty.

[0799] **Simulation**

[0800] Designing and implementing an Internet of Things (IoT) system is a complex task that involves various interconnected components. A proven and common approach to facilitate this process is the utilization of simulations, which play a vital role in achieving efficient, cost-effective, and high-quality IoT systems. Simulations can be employed to emulate (parts of) the end system before actual construction, offering a plethora of benefits and insights for system refinement.

[0801] One of the primary advantages of using simulations is that they provide a detailed understanding of the system's behavior. By replicating the interactions between different components and devices, simulations allow developers and engineers to observe the system's performance and behavior under various scenarios and conditions. This understanding enables them to identify potential issues, inefficiencies, or even critical flaws in the design, thus facilitating improvements before any physical implementation is initiated. Consequently, errors and problems can be detected and addressed at a much earlier stage, reducing the likelihood of costly and time-consuming modifications during the physical construction phase.

[0802] Another area where simulations greatly contribute is in the enhancement of machine learning models. Many IoT systems incorporate machine learning algorithms to process and analyze data collected from various sensors. By feeding simulated data into these algorithms, developers can fine-tune and optimize the models' parameters, leading to more accurate and efficient decision-making capabilities once the IoT system is operational.

[0803] Moreover, simulations aid in calibrating the values for sensors and actuators, typically in a bench top laboratory environment. These physical components are integral to IoT systems, as they gather data from the environment and act upon it. Before actual implementation, simulations allow for the testing and adjustment of sensor accuracy and actuator responsiveness. This calibration ensures that the IoT system will perform optimally and produce reliable results when deployed in real-world scenarios.

[0804] Simulations can take on different forms, depending on the complexity and requirements of the IoT system being developed. A simulation can be purely software-based, where virtual environments and models are created to mimic the behavior of the actual

hardware components. This software-only approach provides a cost-effective way to test and validate the system design before committing resources to hardware manufacturing.

[0805] Alternatively, simulations can encompass a combination of software and hardware components. In such cases, physical prototypes or partial hardware implementations can be integrated with the virtual simulation environment. This approach allows for more accurate representations of the actual system's behavior, enabling engineers to assess interactions between real and virtual elements, identify potential integration challenges, and further improve the overall design.

[0806] Autocoder supports these common simulation pattern in several ways. The data generator operator (Section 5.5) support the insertion of a task and operator that simulates the events generated by a hardware device. Debugging (Section 12) is also a specific form of simulation.

[0807] Simulation Optimization

[0808] Fig. 75 shows an example architecture 7500 of a simulation optimization. The Autocoder system can serve as a kernel component to optimization. There are many forms of optimization which Autocoder can serve as a component. Here we describe simulation optimization where the Autocoder simulation is used to generate data to evaluate the objective function of an optimizer. The simulation optimization contains two layers. The upper layer generates the input to the lower layer (configuration) and receives data from the lower layer (results) to compute the value of an optimization objective function (part of the simulation configuration). Many strategies are possible - in a simple strategy the optimization configuration specifies the dimensions to explore in the configuration space and the optimizer exhaustively explores them.

[0809] As a simple example, consider an application [4] with four operators in a pipeline. The Camera operator generates an image, the Crop operator crops out faces in the image, the Recognize operator recognizes known faces, and a dashboard that lists results. The optimizer objective function is the total bytes communicated through distributed communication. The optimizer configuration specifies that the placement of tasks is the dimension to explore. The optimizer generates a series of configurations of the system with a legal placement. One legal placement, a cloud configuration, puts the Camera operators at doors and all other operators (and associated task-instances) at the root of the entity hierarchy. This configuration is sent to Autocoder and a simulation is performed to measure total bytes communicated. The result of the simulation is sent back to the optimizer. Note that depending on the configuration of Autocoder, hardware may be used in this simulation.

Eventually the optimizer stops and reports the configuration with the lowest value of the optimization function.

[0810] Automatic Application Modification for Debugging

[0811] A common pattern for software debugging isolates a subpart of the application by creating an isolated sandbox. This sandbox has (mostly) repeatable input, allowing the software developer to repeatedly run the sub-application to observe and modify its behavior in a controlled way.

[0812] Autocoder supports this behavior with the Zoom debugging tool. The Zoom tool automatically "carves out" a sub-application of an application. The result is a sandbox, with input and output data, that allows a software engineer the opportunity to explore and modify the behavior of a sub-application in isolation from the result of the application.

[0813] The construction of the sub-application consist of three steps. In the first step, the software developer specifies the target tasks. Fig. 76 shows an example of targeting tasks 7600 for zoom debugging. In the running example, the crop and recognize tasks are chosen. The zoom tool then automatically inserts logging into all inputs to the targets tasks and all outputs, and executes the system, recording (with timestamps) the input and output. Fig. 77 shows an example of logging input and output for zoom debugging 7700. The logged information is recorded into a database. In the third step, the tool constructs a new application containing only the targeted tasks (and associated information) and adds replay and compare operators. The replay operator reads prior logged input and injects it, at the appropriate relative time, into the sub-application. The compare task receives the input from the sub-application and compares it to the generated output from the prior simulation. The sub-application is now ready for the software developer to explore.

[0814] The targeting step consists of a single action: (1) Acquire a set Z of tasks of the application. The tasks in Z are typically connected. The logging step consists of a sequence of actions: (1) Construct a copy of the application. Subsequent actions are performed on the copy. (2) Construct a subgraph G consisting of the tasks in Z and their connections. (3) Insert a logging task for every input of the G from the application and every output to the application. No logging for task input/output within G . (4) Add provenance for any task generating input to an added logging task. (5) Connect the logging tasks to a debugging database (with predefined location). (6) Execute the application given the configuration. The configuration maybe modified by user requests.

[0815] For the set up for the sub-application for further use. (1) For every logging task of an input, replace the logging task with a replay task. (2) For every logging task of an output,

replace the logging task with a compare task. Fig. 78 shows an example 7800 of replaying targeted tasks from logged input and output.

[0816] Fig. 79 shows a block diagram illustrating an example process 7900 for generating a low-code application, according to some implementations of the present disclosure. For clarity of presentation, the description that follows generally describes method 7900 in the context of the other figures in this description. However, it will be understood that method 7900 can be performed, for example, by any suitable system, environment, software, and hardware, or a combination of systems, environments, software, and hardware, as appropriate. In some implementations, various steps of method 7900 can be run in parallel, in combination, in loops, or in any order.

[0817] The process 7900 includes receiving (7902) an application specification defining an application configured for processing data, the data comprising a data type that is specified in the application specification. The process 7900 includes determining (7904), from the application specification, a set of execution modules, each execution module configured for performing a data processing task that is specified in the application specification. The process 7900 includes generating (7906), based on the set of execution modules, a runtime configuration of the application to process the data having the data type.

[0818] Fig. 80 shows a block diagram illustrating an example process 8000 for generating a low-code application, according to some implementations of the present disclosure. For clarity of presentation, the description that follows generally describes method 8000 in the context of the other figures in this description. However, it will be understood that method 8000 can be performed, for example, by any suitable system, environment, software, and hardware, or a combination of systems, environments, software, and hardware, as appropriate. In some implementations, various steps of method 8000 can be run in parallel, in combination, in loops, or in any order.

[0819] The process 8000 includes receiving (8002) receiving an application specification defining an application configured for processing for a specified domain. The process 8000 includes determining (8004), from the application specification, a dataflow graph comprising: a set of operators for processing the data, wherein operators of the set of operators are associated with the specified domain, each operator configured to perform one or more processing steps to perform a function that corresponds to the domain; and a set of links that connect the operators of the set of operators, a link of the set specifying output data from a first operator for being input to a second operator, the output data corresponding to the domain and input data corresponding to the domain. The process 8000 includes generating (8006), based

on the dataflow graph, a runtime configuration of the application to process the data for the specified domain.

[0820] Fig. 81 shows a block diagram illustrating an example process 8100 for generating a low-code application, according to some implementations of the present disclosure. For clarity of presentation, the description that follows generally describes method 8100 in the context of the other figures in this description. However, it will be understood that method 8100 can be performed, for example, by any suitable system, environment, software, and hardware, or a combination of systems, environments, software, and hardware, as appropriate. In some implementations, various steps of method 8100 can be run in parallel, in combination, in loops, or in any order.

[0821] The process 8100 includes receiving (8102) an application specification defining an application configured for data processing to perform a set of functions, the application specification specifying: a set of operators for processing the data, each operator configured to perform one or more processing steps to perform a function of the set of functions; and a set of links that connect the operators of the set of operators, a link of the set specifying output data from a first operator for being input to a second operator. The process 8100 includes generating (8104), based on the set of operators and the set of links, an application instance configured for processing the data by performing operations including: identifying (8106), from an ontology of the application specification, operators of the set of operators and one or more links of the set of links; generating (8108), based on the determining, one or more instances of the operators connected by the one or more links, wherein the one or more instances of the operators are generated based on the identified one or more links of the ontology; and generating (8110) an instance of the application comprising the one or more instances of the operators connected by the one or more links.

[0822] Fig. 82 shows a block diagram illustrating an example process 8200 for executing an application, according to some implementations of the present disclosure. For clarity of presentation, the description that follows generally describes method 8200 in the context of the other figures in this description. However, it will be understood that method 8200 can be performed, for example, by any suitable system, environment, software, and hardware, or a combination of systems, environments, software, and hardware, as appropriate. In some implementations, various steps of method 8200 can be run in parallel, in combination, in loops, or in any order.

[0823] Process 8200 includes initializing (8202) an instance of an application based on the an application specification defining an application configured for data processing to perform a

set of functions, the application specification specifying: a set of operators for processing the data, each operator configured to perform one or more processing steps to perform a function of the set of functions; a set of links that connect the operators of the set of operators, a link of the set specifying output data from a first operator for being input to a second operator; and a set of task instances specifying functions of the set of functions for completion by the application instance. Initializing the instance of the application comprises: for each task instance, generating (8204) an instance of an operator associated with the task instance and associating the instance of the operator with one or more outgoing links. Process 8200 includes executing (8206) the instance of the application by performing operations comprising: for each generated instance of an operator, causing the operator to wait for a trigger condition; and responsive to satisfaction of the trigger condition, causing the generated instance of the operator to perform an associated function.

[0824] Fig. 83 shows a block diagram illustrating an example process 8300 for generating a low-code application, according to some implementations of the present disclosure. For clarity of presentation, the description that follows generally describes method 8300 in the context of the other figures in this description. However, it will be understood that method 8300 can be performed, for example, by any suitable system, environment, software, and hardware, or a combination of systems, environments, software, and hardware, as appropriate. In some implementations, various steps of method 8300 can be run in parallel, in combination, in loops, or in any order.

[0825] Process 8300 includes receiving (8302) an application specification defining an application configured for processing a data stream, the data stream comprising a data type that is specified in the application specification. Process 8300 includes determining (8304), from the application specification, a set of execution modules, each execution module configured for performing a data processing task that is specified in the application specification. Process 8306 includes generating, based on the set of execution modules, a runtime configuration of the application to process the data stream having the data type.

[0826] Fig. 84 is a block diagram of an example computer system 8400 used to provide computational functionalities associated with described algorithms, methods, functions, processes, flows, and procedures described in the present disclosure (such as the methods described previously with reference to Figs. 79-83), according to some implementations of the present disclosure. The illustrated computer 1002 is intended to encompass any computing device such as a server, a desktop computer, a laptop/notebook computer, a wireless data port, a smart phone, a personal data assistant (PDA), a tablet computing device, or one or more

processors within these devices, including physical instances, virtual instances, or both. The computer 1002 can include input devices such as keypads, keyboards, and touch screens that can accept user information. Also, the computer 1002 can include output devices that can convey information associated with the operation of the computer 1002. The information can include digital data, visual data, audio information, or a combination of information. The information can be presented in a graphical user interface (UI or GUI).

[0827] The computer 1002 can serve in a role as a client, a network component, a server, a database, a persistency, or components of a computer system for performing the subject matter described in the present disclosure. The illustrated computer 1002 is communicably coupled with a network 1030. In some implementations, one or more components of the computer 1002 can be configured to operate within different environments, including cloud-computing-based environments, local environments, global environments, and combinations of environments.

[0828] At a high level, the computer 1002 is an electronic computing device operable to receive, transmit, process, store, and manage data and information associated with the described subject matter. According to some implementations, the computer 1002 can also include, or be communicably coupled with, an application server, an email server, a web server, a caching server, a streaming data server, or a combination of servers.

[0829] The computer 1002 can receive requests over network 1030 from a client application (for example, executing on another computer 1002). The computer 1002 can respond to the received requests by processing the received requests using software applications. Requests can also be sent to the computer 1002 from internal users (for example, from a command console), external (or third) parties, automated applications, entities, individuals, systems, and computers.

[0830] Each of the components of the computer 1002 can communicate using a system bus 1003. In some implementations, any or all of the components of the computer 1002, including hardware or software components, can interface with each other or the interface 1004 (or a combination of both), over the system bus 1003. Interfaces can use an application programming interface (API) 1012, a service layer 1013, or a combination of the API 1012 and service layer 1013. The API 1012 can include specifications for routines, data structures, and object classes. The API 1012 can be either computer-language independent or dependent. The API 1012 can refer to a complete interface, a single function, or a set of APIs.

[0831] The service layer 1013 can provide software services to the computer 1002 and other components (whether illustrated or not) that are communicably coupled to the computer 1002. The functionality of the computer 1002 can be accessible for all service consumers using

this service layer. Software services, such as those provided by the service layer 1013, can provide reusable, defined functionalities through a defined interface. For example, the interface can be software written in JAVA, C++, or a language providing data in extensible markup language (XML) format. While illustrated as an integrated component of the computer 1002, in alternative implementations, the API 1012 or the service layer 1013 can be stand-alone components in relation to other components of the computer 1002 and other components communicably coupled to the computer 1002. Moreover, any or all parts of the API 1012 or the service layer 1013 can be implemented as child or sub-modules of another software module, enterprise application, or hardware module without departing from the scope of the present disclosure.

[0832] The computer 1002 includes an interface 1004. Although illustrated as a single interface 1004 in Fig. 85, two or more interfaces 1004 can be used according to particular needs, desires, or particular implementations of the computer 1002 and the described functionality. The interface 1004 can be used by the computer 1002 for communicating with other systems that are connected to the network 1030 (whether illustrated or not) in a distributed environment. Generally, the interface 1004 can include, or be implemented using, logic encoded in software or hardware (or a combination of software and hardware) operable to communicate with the network 1030. More specifically, the interface 1004 can include software supporting one or more communication protocols associated with communications. As such, the network 1030 or the interface's hardware can be operable to communicate physical signals within and outside of the illustrated computer 1002.

[0833] The computer 1002 includes a processor 1005. Although illustrated as a single processor 1005 in FIG. 10, two or more processors 1005 can be used according to particular needs, desires, or particular implementations of the computer 1002 and the described functionality. Generally, the processor 1005 can execute instructions and can manipulate data to perform the operations of the computer 1002, including operations using algorithms, methods, functions, processes, flows, and procedures as described in the present disclosure.

[0834] The computer 1002 also includes a database 1006 that can hold data for the computer 1002 and other components connected to the network 1030 (whether illustrated or not). For example, database 1006 can be an in-memory, conventional, or a database storing data consistent with the present disclosure. In some implementations, database 1006 can be a combination of two or more different database types (for example, hybrid in-memory and conventional databases) according to particular needs, desires, or particular implementations of the computer 1002 and the described functionality. Although illustrated as a single database

1006 in FIG. 10, two or more databases (of the same, different, or combination of types) can be used according to particular needs, desires, or particular implementations of the computer 1002 and the described functionality. While database 1006 is illustrated as an internal component of the computer 1002, in alternative implementations, database 1006 can be external to the computer 1002.

[0835] The computer 1002 also includes a memory 1007 that can hold data for the computer 1002 or a combination of components connected to the network 1030 (whether illustrated or not). Memory 1007 can store any data consistent with the present disclosure. In some implementations, memory 1007 can be a combination of two or more different types of memory (for example, a combination of semiconductor and magnetic storage) according to particular needs, desires, or particular implementations of the computer 1002 and the described functionality. Although illustrated as a single memory 1007 in FIG. 10, two or more memories 1007 (of the same, different, or combination of types) can be used according to particular needs, desires, or particular implementations of the computer 1002 and the described functionality. While memory 1007 is illustrated as an internal component of the computer 1002, in alternative implementations, memory 1007 can be external to the computer 1002.

[0836] The application 1008 can be an algorithmic software engine providing functionality according to particular needs, desires, or particular implementations of the computer 1002 and the described functionality. For example, application 1008 can serve as one or more components, modules, or applications. Further, although illustrated as a single application 1008, the application 1008 can be implemented as multiple applications 1008 on the computer 1002. In addition, although illustrated as internal to the computer 1002, in alternative implementations, the application 1008 can be external to the computer 1002.

[0837] The computer 1002 can also include a power supply 1014. The power supply 1014 can include a rechargeable or non-rechargeable battery that can be configured to be either user- or non-user-replaceable. In some implementations, the power supply 1014 can include power-conversion and management circuits, including recharging, standby, and power management functionalities. In some implementations, the power-supply 1014 can include a power plug to allow the computer 1002 to be plugged into a wall socket or a power source to, for example, power the computer 1002 or recharge a rechargeable battery.

[0838] There can be any number of computers 1002 associated with, or external to, a computer system containing computer 1002, with each computer 1002 communicating over network 1030. Further, the terms "client," "user," and other appropriate terminology can be used interchangeably, as appropriate, without departing from the scope of the present

disclosure. Moreover, the present disclosure contemplates that many users can use one computer 1002 and one user can use multiple computers 1002.

[0839] While this specification contains many specific implementation details, these should not be construed as limitations on the scope of what may be claimed, but rather as descriptions of features that may be specific to particular implementations. Certain features that are described in this specification in the context of separate implementations can also be implemented, in combination, in a single implementation. Conversely, various features that are described in the context of a single implementation can also be implemented in multiple implementations, separately, or in any suitable sub-combination. Moreover, although previously described features may be described as acting in certain combinations and even initially claimed as such, one or more features from a claimed combination can, in some cases, be excised from the combination, and the claimed combination may be directed to a sub-combination or variation of a sub-combination.

[0840] In the foregoing description, embodiments of the invention have been described with reference to numerous specific details that may vary from implementation to implementation. The description and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense. The sole and exclusive indicator of the scope of the invention, and what is intended by the applicants to be the scope of the invention, is the literal and equivalent scope of the set of claims that issue from this application, in the specific form in which such claims issue, including any subsequent correction. Any definitions expressly set forth herein for terms contained in such claims shall govern the meaning of such terms as used in the claims. In addition, when we use the term “further comprising” or “further including” in the foregoing description or following claims, what follows this phrase can be an additional step or entity, or a sub-step/sub-entity of a previously-recited step or entity.

[0841] Particular implementations of the subject matter have been described. Other implementations, alterations, and permutations of the described implementations are within the scope of the following claims as are apparent to those skilled in the art. While operations are depicted in the drawings or claims in a particular order, this should not be understood as requiring that such operations be performed in the particular order shown or in sequential order, or that all illustrated operations be performed (some operations may be considered optional), to achieve desirable results. In certain circumstances, multitasking or parallel processing (or a combination of multitasking and parallel processing) may be advantageous and performed as deemed appropriate.

[0842] Moreover, the separation or integration of various system modules and components in the previously described implementations should not be understood as requiring such separation or integration in all implementations, and it should be understood that the described program components and systems can generally be integrated together in a single software product or packaged into multiple software products.

[0843] Accordingly, the previously described example implementations do not define or constrain the present disclosure. Other changes, substitutions, and alterations are also possible without departing from the spirit and scope of the present disclosure.

[0844] Furthermore, any claimed implementation is considered to be applicable to at least a computer-implemented method; a non-transitory, computer-readable medium storing computer-readable instructions to perform the computer-implemented method; and a computer system comprising a computer memory interoperably coupled with a hardware processor configured to perform the computer-implemented method or the instructions stored on the non-transitory, computer-readable medium.

[0845] Note that in a distributed system any data sent or received by a task instance via a notification or call might cross a process boundary. Thus Autocoder requires all message payloads to be serializable. Not knowing in advance how the processing will be split and which messages will cross process boundaries implies that all payloads can potentially be required to be serialized.

[0846] The requirement of payload data serialization for all payload message data types imposes a coding burden on the engineer. In keeping with our low-code design objective, Autocoder offers two alternatives to implement serialization for subclasses of the Payload class.

[0847] In the first alternative, the Payload class itself includes a collection of utility functions that automatically provide serialization/deserialization functionality, as long as the engineer utilizes typical data structures (strings, numbers, arrays, dictionaries, images, etc.). The current implementation uses JSON as the serialization format but obviously any other format (e.g., XML) is equally good. The utility functions generate data that is self-describing. Self-describing data simplifies the build and deployment process. This automatic functionality is almost always used in practice.

[0848] The second alternative is for the engineer to customize the implementation of serialization and deserialization methods. The customization must conform to the Autocoder interface. If the customized methods are self-describing, then no additional modifications are

necessary. The custom implementation is then automatically invoked by Autocoder during runtime.

[0849] Note that errors that occur during the execution of an operator (and/or need to be sent as result of a call) also need to be serialized/deserialized correctly. For this purpose, the Payload class has a special subclass ErrorPayload. Each error message sent as a potential response to a call is a subclass of this class, which ensures correct transport of the message content.

[0850] A number of embodiments of these systems and methods have been described. Nevertheless, it is understood that various modifications may be made without departing from the spirit and scope of this disclosure.

[0851] Examples

[0852] The embodiments described herein enable one or more of the following examples or embodiments.

[0853] Example 1 includes a method for generating a low-code application includes receiving an application specification defining an application configured for processing data, the data comprising a data type that is specified in the application specification; determining, from the application specification, a set of execution modules, each execution module configured for performing a data processing task that is specified in the application specification; and generating, based on the set of execution modules, a runtime configuration of the application to process the data having the data type.

[0854] Example 2 may include the method of example 1, wherein the application specification defining an entity ontology that specifies entities representing real-world concepts and one or more relationships between the entities.

[0855] Example 3 may include the method of any of examples 1-2, wherein the one or more relationships are semantic relationships representing conceptual relationships between the entities.

[0856] Example 4 may include the method of any of examples 1-3, wherein the one or more relationships are logical relationships between the entities.

[0857] Example 5 may include the method of any of examples 1-4, wherein the entity ontology comprises a graph, wherein nodes of the graph represent the entities and wherein edges of the graph represent the one or more relationships between the entities.

[0858] Example 6 may include the method of any of examples 1-5, the application specification defining a library of data types that are able to be processed by the application.

[0859] Example 7 may include the method of any of examples 1-6, wherein the library of data types comprises a database schema of application domain data, the application domain data including definitions for entities of the domain and definitions for relationships between the entities.

[0860] Example 8 may include the method of any of examples 1-7, wherein a data type comprises a semantic meaning of an entity.

[0861] Example 9 may include the method of any of examples 1-8, wherein a data type comprises a data protocol, a data format, a data standard, or a data specification that defines what data having the data type represents.

[0862] Example 10 may include the method of any of examples 1-9, wherein the library of data types comprises a set of entities.

[0863] Example 11 may include the method of any of examples 1-10, wherein each the library of data types associates one or more data types with one or more valid operators for processing data having the data type.

[0864] Example 12 may include the method of any of examples 1-11, the application specification defining an operations algebra that specifies a set of operators, each operator associated with a code component that is available for execution by the application, wherein each code component is configured to perform a pre-defined operation.

[0865] Example 13 may include the method of any of examples 1-12, wherein a code component of the set of code components is configured to be a stand-alone and reusable code subset for performing the pre-defined operation for one or more domains including at least one domain specified by the application specification.

[0866] Example 14 may include the method of any of examples 1-13, wherein a code component comprises one of an image processing model, machine learning logic, a data enrichment model, or a data automata.

[0867] Example 15 may include the method of any of examples 1-14, wherein a code component comprises a set of logical instructions that are defined by one or more parameters.

[0868] Example 16 may include the method of any of examples 1-15, wherein values of one or more the parameters for the code component are domain-independent.

[0869] Example 17 may include the method of any of examples 1-16, wherein values of the one or more parameters for the code component are domain-specific, and wherein the code component is combined with at least another code component defined by one or more parameters that are domain-independent.

[0870] Example 18 may include the method of any of examples 1-17, wherein each operator of the operations algebra comprises an atomic computational block with at least one pre-defined input and at least one pre-defined output.

[0871] Example 19 may include the method of any of examples 1-18, wherein each code component is associated with a respective programming language type for performing the operations.

[0872] Example 20 may include the method of any of examples 1-20, wherein a first code component is associated with a first programming language type, and wherein a second code component is associated with a second programming language type that is different from the first programming language type.

[0873] Example 21 may include the method of any of examples 1-20, wherein the pre-defined operation comprises one or more processing steps to perform a function.

[0874] Example 22 may include the method of any of examples 1-21, wherein an operator is configured to be executed by the application responsive to the application receiving input data comprising a process notification that is an asynchronous input event.

[0875] Example 23 may include the method of any of examples 1-22, wherein an operator is configured to be executed by the application responsive to the application receiving input data comprising a compute request that is a synchronous input event requesting immediate computation and return of a result.

[0876] Example 24 may include the method of any of examples 1-23, wherein an operator is configured to generate output data comprising a send notification for triggering another code component, the send notification being asynchronous.

[0877] Example 25 may include the method of any of examples 1-24, wherein an operator is configured to generate output data comprising a call request requesting immediate computation and return of a result by another operator, the call request being synchronous.

[0878] Example 26 may include the method of any of examples 1-25, wherein an operator is configured to generate output data comprising a call request requesting immediate computation and return of a result by another operator, the call request being synchronous.

[0879] Example 27 may include the method of any of examples 1-26, wherein an operator comprises an analog component.

[0880] Example 28 may include the method of any of examples 1-27, wherein the analog component is configured to be executed by the application responsive to the application receiving input data comprising one or more of a hardware signal, a system signal, an external software trigger, and an external network call.

[0881] Example 29 may include the method of any of examples 1-28, wherein an operator comprises a sensor.

[0882] Example 30 may include the method of any of examples 1-29, wherein an operator comprises a machine learning model trained to perform the pre-defined operation.

[0883] Example 31 may include the method of any of examples 1-30, wherein an operator comprises a machine learning model trained using domain-specific training data.

[0884] Example 32 may include the method of any of examples 1-31, wherein an operator comprises a data interface.

[0885] Example 33 includes a method for generating an application comprising a dataflow graph, the method comprising: receiving an application specification defining an application configured for processing for a specified domain, determining, from the application specification, a dataflow graph comprising: a set of operators for processing the data, wherein

operators of the set of operators are associated with the specified domain, each operator configured to perform one or more processing steps to perform a function that corresponds to the domain; and a set of links that connect the operators of the set of operators, a link of the set specifying output data from a first operator for being input to a second operator, the output data corresponding to the domain and input data corresponding to the domain; generating, based on the dataflow graph, a runtime configuration of the application to process the data for the specified domain.

[0886] Example 34 may include the method of example 33, wherein the domain is specified by setting a value of a parameter for generating the application specification.

[0887] Example 35 may include the method of any of examples 33-34, wherein an operator performs the function based on a type of the domain.

[0888] Example 36 may include the method of any of examples 33-35, wherein a link of the set of links represents dataflow between at least two operators.

[0889] Example 37 may include the method of any of examples 33-36, wherein an operator is associated with a set of task definitions that control execution of the processing steps of the operator to perform the function.

[0890] Example 38 may include the method of any of examples 33-37, wherein the task definitions are based on parameter values for generating the application specification.

[0891] Example 39 may include the method of any of examples 33-38, further comprising associating the set of operators with an ontology defined in the application specification, the ontology comprising a mapping of one or more task definitions for processing the data to an operator, the operator configured to send processed data based on the one or more task definitions to one or more other operators connected to the operator by a link of the set of links.

[0892] Example 40 may include the method of any of examples 33-39, wherein at last one task definition of the one or more task definition is mapped to a plurality of operators.

[0893] Example 41 may include the method of any of examples 33-40, further comprising associating the dataflow graph with a function scope for each operator of the dataflow graph, the function scope mapping one or more tasks to each operator of the set of operators.

[0894] Example 42 may include the method of any of examples 33-41, further comprising associating the dataflow graph with a function relationship for each link of the dataflow graph, the function relationship mapping each link in the set of links to a respective relationship defining in an ontology including the set of operators.

[0895] Example 43 may include a method for generating an application comprising a dataflow graph, the method comprising: receiving an application specification defining an

application configured for data processing to perform a set of functions, the application specification specifying: a set of operators for processing the data, each operator configured to perform one or more processing steps to perform a function of the set of functions; and a set of links that connect the operators of the set of operators, a link of the set specifying output data from a first operator for being input to a second operator; generating, based on the set of operators and the set of links, an application instance configured for processing the data by performing operations including: identifying, from an ontology of the application specification, operators of the set of operators and one or more links of the set of links; generating, based on the determining, one or more instances of the operators connected by the one or more links, wherein the one or more instances of the operators are generated based on the identified one or more links of the ontology; and generating an instance of the application comprising the one or more instances of the operators connected by the one or more links.

[0896] Example 44 may include the method of example 43, wherein the ontology specifies a type of link, and wherein the one or more instances of the operators are generated when the one or more instances of the operators are connected by a link having the type.

[0897] Example 45 may include the method of any of examples 43-44, wherein the identified operators are a subset of the set of operators, and wherein the identified one or more links are a subset of the set of links.

[0898] Example 46 may include the method of any of examples 43-45, wherein the ontology of the application species one or more relationships between entities represented by the operators.

[0899] Example 47 may include the method of any of examples 43-46, wherein the relationships correspond to domain-specific relationships specified in the application specification.

[0900] Example 48 may include the method of any of examples 43-47, wherein the domain-specific relationships correspond to relationships between real-world objects.

[0901] Example 49 may include the method of any of examples 43-48, further comprising generating, based on the instance of the application, runtime logic to process the data.

[0902] Example 50 may include the method of any of examples 43-49, wherein the runtime logic comprises a runtime graph including the one or more instances of the operators connected by the one or more links.

[0903] Example 51 may include the method of any of examples 43-50, wherein each of the one or more instances of the operators is configured to: maintain a local processing thread; perform a function autonomously from one or more other instances of the operators; and

communicate using the one or more links with the one or more other instances of the operators using messages.

[0904] Example 52 may include the method of any of examples 43-51, wherein the messages are configured based on a domain of the application specification.

[0905] Example 53 may include a method for executing an application, the method comprising: initializing an instance of an application based on the an application specification defining an application configured for data processing to perform a set of functions, the application specification specifying: a set of operators for processing the data, each operator configured to perform one or more processing steps to perform a function of the set of functions; a set of links that connect the operators of the set of operators, a link of the set specifying output data from a first operator for being input to a second operator; and a set of task instances specifying functions of the set of functions for completion by the application instance; wherein initializing the instance of the application comprises: for each task instance, generating an instance of an operator associated with the task instance; associating the instance of the operator with one or more outgoing links; and executing the instance of the application by performing operations comprising: for each generated instance of an operator, causing the operator to wait for a trigger condition; responsive to satisfaction of the trigger condition, causing the generated instance of the operator to perform an associated function.

[0906] Example 54 may include the method of example 53, wherein the trigger condition comprises receipt of a message at the generated instance of the operator over an input link to the generated instance of the operator.

[0907] Example 55 may include the method of any of examples 53-54, wherein the message is received at the generated instance of the operator at an input port of the operator.

[0908] Example 56 may include the method of any of examples 53-55, wherein the receipt of the message at the input port of the operator causes a processor to perform an operation and generate an output message for transmission from an output port of the operator.

[0909] Example 57 may include the method of any of examples 53-56, wherein the output message is transmitted asynchronously to each operator instance connected to the output port.

[0910] Example 58 may include the method of any of examples 53-57, wherein the receipt of the message at the input port of the operator causes a processor to perform an operation and generate a message to send on call request ports for synchronous processing.

[0911] Example 59 may include the method of any of examples 53-58, wherein the message is immutable.

[0912] Example 60 may include the method of any of examples 53-59, wherein each instance of an operator maintains an internal state during execution of the application.

[0913] Example 61 may include the method of any of examples 53-60, wherein the internal state of each instance of an operator is independent from internal states of other instances of operators.

[0914] Example 62 may include a method for generating a low-code application, the method comprising: receiving an application specification defining an application configured for processing a data stream, the data stream comprising a data type that is specified in the application specification; determining, from the application specification, a set of execution modules, each execution module configured for performing a data processing task that is specified in the application specification; and generating, based on the set of execution modules, a runtime configuration of the application to process the data stream having the data type.

[0915] Example 63 may include the method of example 62, the method further comprising: receiving a distributed execution plan that is part of the application specification; and configuring, in accordance with the distributed execution plan, the runtime configuration of the application for distributed execution.

[0916] Example 64 may include the method of any of examples 62-63, the method further comprising: based on the distributed execution plan determine a cost estimate for execution of the application.

[0917] Example 65 may include the method of any of examples 62-64, the method further comprising: receiving, as part of the application specification, one or more domain constraints of the runtime configuration of the application; and generating, based on the one or more domain constraints, an execution plan for execution of the runtime configuration of the application, the execution plan satisfying the one or more domain constraints.

[0918] Example 66 may include the method of any of examples 62-65, the method further comprising: optimizing, based on the execution plan, distributed execution of the runtime configuration of the application, the optimization satisfying the one or more domain constraints.

[0919] Example 67 may include the method of any of examples 62-66, the method further comprising: generating a unit test for testing the runtime configuration of the application, the unit test configured to identify data leakage or at least one fault in one or more of the execution module when the fault occurs.

[0920] Example 68 may include the method of any of examples 62-69, the method further comprising receiving a security requirement as part of the application specification; generating a security execution module for including in the runtime configuration of the application, the security execution module satisfying the security requirement of the application specification; and generating the runtime configuration of the application including the security execution module.

[0921] Example 69 may include the method of any of examples 62-68, the method further comprising receiving update data specifying a change to the application specification, the update data specifying at least one data processing action; and updating, automatically, the runtime configuration of the application to perform the at least one data processing action.

[0922] Example 70 may include the method of any of examples 62-69, wherein the update data specify one or more of error management logic, synchronization logic, logging logic, calibration logic, control logic, and user visualization logic.

[0923] A data processing system comprising at least one processor and a memory storing instructions configured to cause, when executed by the at least one processor, the at least one processor to perform any of the operations of claims 1-70.

[0924] One or more non-transitory computer readable media storing instructions, that, when executed by at least one processor, are configured to cause the at least one processor to perform any of the operations of claims 1-70.

WHAT IS CLAIMED IS:

1. A method for generating a low-code application, the method comprising:
receiving an application specification defining an application configured for processing data, the data comprising a data type that is specified in the application specification;
determining, from the application specification, a set of execution modules, each execution module configured for performing a data processing task that is specified in the application specification; and
generating, based on the set of execution modules, a runtime configuration of the application to process the data having the data type.
2. The method of claim 1, the application specification defining an entity ontology that specifies entities representing real-world concepts and one or more relationships between the entities.
3. The method of claim 2, wherein the one or more relationships are semantic relationships representing conceptual relationships between the entities.
4. The method of claim 2, wherein the one or more relationships are logical relationships between the entities.
5. The method of claim 2, wherein the entity ontology comprises a graph, wherein nodes of the graph represent the entities and wherein edges of the graph represent the one or more relationships between the entities.
6. The method of claim 1, the application specification defining a library of data types that are able to be processed by the application.
7. The method of claim 6, wherein the library of data types comprises a database schema of application domain data, the application domain data including definitions for entities of the domain and definitions for relationships between the entities.
8. The method of claim 6, wherein a data type comprises a semantic meaning of an entity.

9. The method of claim 6, wherein a data type comprises a data protocol, a data format, a data standard, or a data specification that defines what data having the data type represents.
10. The method of claim 6, wherein the library of data types comprises a set of entities.
11. The method of claim 6, wherein each the library of data types associates one or more data types with one or more valid operators for processing data having the data type.
12. The method of claim 1, the application specification defining an operations algebra that specifies a set of operators, each operator associated with a code component that is available for execution by the application, wherein each code component is configured to perform a pre-defined operation.
13. The method of claim 12, wherein a code component of the set of code components is configured to be a stand-alone and reusable code subset for performing the pre-defined operation for one or more domains including at least one domain specified by the application specification.
14. The method of claim 12, wherein a code component comprises one of an image processing model, machine learning logic, a data enrichment model, or a data automata.
15. The method of claim 12, wherein a code component comprises a set of logical instructions that are defined by one or more parameters.
16. The method of claim 15, wherein values of one or more the parameters for the code component are domain-independent.
17. The method of claim 15, wherein values of the one or more parameters for the code component are domain-specific, and wherein the code component is combined with at least another code component defined by one or more parameters that are domain-independent.

18. The method of claim 12, wherein each operator of the operations algebra comprises an atomic computational block with at least one pre-defined input and at least one pre-defined output.

19. The method of claim 12, wherein each code component is associated with a respective programming language type for performing the operations.

20. The method of claim 19, wherein a first code component is associated with a first programming language type, and wherein a second code component is associated with a second programming language type that is different from the first programming language type.

21. The method of claim 12, wherein the pre-defined operation comprises one or more processing steps to perform a function.

22. The method of claim 12, wherein an operator is configured to be executed by the application responsive to the application receiving input data comprising a process notification that is an asynchronous input event.

23. The method of claim 12, wherein an operator is configured to be executed by the application responsive to the application receiving input data comprising a compute request that is a synchronous input event requesting immediate computation and return of a result.

24. The method of claim 12, wherein an operator is configured to generate output data comprising a send notification for triggering another code component, the send notification being asynchronous.

25. The method of claim 12, wherein an operator is configured to generate output data comprising a call request requesting immediate computation and return of a result by another operator, the call request being synchronous.

26. The method of claim 12, wherein an operator is configured to generate output data comprising a call request requesting immediate computation and return of a result by another operator, the call request being synchronous.
27. The method of claim 12, wherein an operator comprises an analog component.
28. The method of claim 27, wherein the analog component is configured to be executed by the application responsive to the application receiving input data comprising one or more of a hardware signal, a system signal, an external software trigger, and an external network call.
29. The method of claim 12, wherein an operator comprises a sensor.
30. The method of claim 12, wherein an operator comprises a machine learning model trained to perform the pre-defined operation.
31. The method of claim 12, wherein an operator comprises a machine learning model trained using domain-specific training data.
32. The method of claim 12, wherein an operator comprises a data interface.
33. A method for generating an application comprising a dataflow graph, the method comprising:
- receiving an application specification defining an application configured for processing for a specified domain;
 - determining, from the application specification, a dataflow graph comprising:
 - a set of operators for processing the data, wherein operators of the set of operators are associated with the specified domain, each operator configured to perform one or more processing steps to perform a function that corresponds to the domain; and
 - a set of links that connect the operators of the set of operators, a link of the set specifying output data from a first operator for being input to a second operator, the output data corresponding to the domain and input data corresponding to the domain;

generating, based on the dataflow graph, a runtime configuration of the application to process the data for the specified domain.

34. The method of claim 33, wherein the domain is specified by setting a value of a parameter for generating the application specification.

35. The method of claim 33, wherein an operator performs the function based on a type of the domain.

36. The method of claim 33, wherein a link of the set of links represents dataflow between at least two operators.

37. The method of claim 33, wherein an operator is associated with a set of task definitions that control execution of the processing steps of the operator to perform the function.

38. The method of claim 37, wherein the task definitions are based on parameter values for generating the application specification.

39. The method of claim 33, further comprising associating the set of operators with an ontology defined in the application specification, the ontology comprising a mapping of one or more task definitions for processing the data to an operator, the operator configured to send processed data based on the one or more task definitions to one or more other operators connected to the operator by a link of the set of links.

40. The method of claim 39, wherein at least one task definition of the one or more task definition is mapped to a plurality of operators.

41. The method of claim 33, further comprising associating the dataflow graph with a function scope for each operator of the dataflow graph, the function scope mapping one or more tasks to each operator of the set of operators.

42. The method of claim 33, further comprising associating the dataflow graph with a function relationship for each link of the dataflow graph, the function relationship

mapping each link in the set of links to a respective relationship defining in an ontology including the set of operators.

43. A method for generating an application comprising a dataflow graph, the method comprising:

receiving an application specification defining an application configured for data processing to perform a set of functions, the application specification specifying:

a set of operators for processing the data, each operator configured to perform one or more processing steps to perform a function of the set of functions; and

a set of links that connect the operators of the set of operators, a link of the set specifying output data from a first operator for being input to a second operator;

generating, based on the set of operators and the set of links, an application instance configured for processing the data by performing operations including:

identifying, from an ontology of the application specification, operators of the set of operators and one or more links of the set of links;

generating, based on the determining, one or more instances of the operators connected by the one or more links, wherein the one or more instances of the operators are generated based on the identified one or more links of the ontology; and

generating an instance of the application comprising the one or more instances of the operators connected by the one or more links.

44. The method of claim 43, wherein the ontology specifies a type of link, and wherein the one or more instances of the operators are generated when the one or more instances of the operators are connected by a link having the type.

45. The method of claim 43, wherein the identified operators are a subset of the set of operators, and wherein the identified one or more links are a subset of the set of links.

46. The method of claim 43, wherein the ontology of the application species one or more relationships between entities represented by the operators.

47. The method of claim 46, wherein the relationships correspond to domain-specific relationships specified in the application specification.

48. The method of claim 47, wherein the domain-specific relationships correspond to relationships between real-world objects.

49. The method of claim 43, further comprising generating, based on the instance of the application, runtime logic to process the data.

50. The method of claim 49, wherein the runtime logic comprises a runtime graph including the one or more instances of the operators connected by the one or more links.

51. The method of claim 43, wherein each of the one or more instances of the operators is configured to:

- maintain a local processing thread;
- perform a function autonomously from one or more other instances of the operators; and
- communicate using the one or more links with the one or more other instances of the operators using messages.

52. The method of claim 51, wherein the messages are configured based on a domain of the application specification.

53. A method for executing an application, the method comprising:
initializing an instance of an application based on the an application specification
defining an application configured for data processing to perform a set of functions, the application specification specifying:

- a set of operators for processing the data, each operator configured to perform one or more processing steps to perform a function of the set of functions;
- a set of links that connect the operators of the set of operators, a link of the set specifying output data from a first operator for being input to a second operator; and
- a set of task instances specifying functions of the set of functions for completion by the application instance;

wherein initializing the instance of the application comprises:

- for each task instance, generating an instance of an operator associated with the task instance;
- associating the instance of the operator with one or more outgoing links; and

executing the instance of the application by performing operations comprising:
for each generated instance of an operator, causing the operator to wait
for a trigger condition; and
responsive to satisfaction of the trigger condition, causing the
generated instance of the operator to perform an associated function.

54. The method of claim 53, wherein the trigger condition comprises receipt of a message at the generated instance of the operator over an input link to the generated instance of the operator.

55. The method of claim 54, wherein the message is received at the generated instance of the operator at an input port of the operator.

56. The method of claim 55, wherein the receipt of the message at the input port of the operator causes a processor to perform an operation and generate an output message for transmission from an output port of the operator.

57. The method of claim 56, wherein the output message is transmitted asynchronously to each operator instance connected to the output port.

58. The method of claim 55, wherein the receipt of the message at the input port of the operator causes a processor to perform an operation and generate a message to send on call request ports for synchronous processing.

59. The method of claim 55, wherein the message is immutable.

60. The method of claim 53, wherein each instance of an operator maintains an internal state during execution of the application.

61. The method of claim 60, wherein the internal state of each instance of an operator is independent from internal states of other instances of operators.

62. A method for generating a low-code application, the method comprising:

receiving an application specification defining an application configured for processing a data stream, the data stream comprising a data type that is specified in the application specification;

determining, from the application specification, a set of execution modules, each execution module configured for performing a data processing task that is specified in the application specification; and

generating, based on the set of execution modules, a runtime configuration of the application to process the data stream having the data type.

63. The method of claim 62, further comprising:

receiving a distributed execution plan that is part of the application specification; and
configuring, in accordance with the distributed execution plan, the runtime configuration of the application for distributed execution.

64. The method of claim 63, further comprising:

based on the distributed execution plan determine a cost estimate for execution of the application.

65. The method of claim 62, further comprising:

receiving, as part of the application specification, one or more domain constraints of the runtime configuration of the application; and

generating, based on the one or more domain constraints, an execution plan for execution of the runtime configuration of the application, the execution plan satisfying the one or more domain constraints.

66. The method of claim 65, further comprising:

optimizing, based on the execution plan, distributed execution of the runtime configuration of the application, the optimization satisfying the one or more domain constraints.

67. The method of claim 62, further comprising:

generating a unit test for testing the runtime configuration of the application, the unit test configured to identify data leakage or at least one fault in one or more of the execution module when the fault occurs.

68. The method of claim 62, further comprising:
receiving a security requirement as part of the application specification;
generating a security execution module for including in the runtime configuration of the application, the security execution module satisfying the security requirement of the application specification; and
generating the runtime configuration of the application including the security execution module.
69. The method of claim 62, further comprising:
receiving update data specifying a change to the application specification, the update data specifying at least one data processing action; and
updating, automatically, the runtime configuration of the application to perform the at least one data processing action.
70. The method of claim 8, wherein the update data specify one or more of error management logic, synchronization logic, logging logic, calibration logic, control logic, and user visualization logic.
71. A data processing system comprising at least one processor and a memory storing instructions configured to cause, when executed by the at least one processor, the at least one processor to perform any of the operations of claims 1-70.
72. One or more non-transitory computer readable media storing instructions, that, when executed by at least one processor, are configured to cause the at least one processor to perform any of the operations of claims 1-70.