

(51) International Patent Classification:
G06F 19/00 (2011.01)(21) International Application Number:
PCT/US2016/033924(22) International Filing Date:
24 May 2016 (24.05.2016)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
62/166,927 27 May 2015 (27.05.2015) US(71) Applicant: **BOARD OF REGENTS, THE UNIVERSITY OF TEXAS SYSTEM** [US/US]; 201 West 7th Street, Austin, TX 78701 (US).(72) Inventors: **SENTIS, Luis**; 204 E. Dean Keeton Street, Stop C2200, Austin, TX 78712-1591 (US). **FOK, Chi-en-Liang**; 204 E. Dean Keeton Street, Stop C2200, Austin, TX 78712-1591 (US). **JOHNSON, Gwendolyn**; 204 E. Dean Keeton Street, Stop C2200, Austin, TX 78712-1591(US). **YAMOKOSKI, John, D.**; 204 E. Dean Keeton Street, Stop C2200, Austin, TX 78712-1591 (US).(74) Agent: **VOIGT, Robert, A., Jr.**; Winstead, P.C., P.O. Box 131851, Dallas, TX 75313 (US).

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

(84) Designated States (unless otherwise indicated, for every kind of regional protection available): ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU,

[Continued on next page]

(54) Title: INTEGRATION OF WHOLE BODY CONTROLLERS WITH ROBOTS

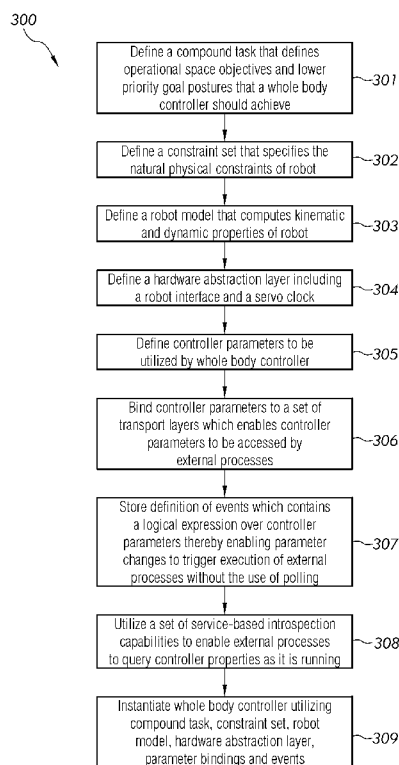


FIG. 3

(57) Abstract: A method, system and computer program product for controlling a humanoid robot. A compound task that defines operational space objectives and lower priority goal postures that a whole body controller should achieve is defined. Furthermore, a constraint set that specifies natural physical constraints of a robot is defined. Additionally, a robot model that computes kinematic and dynamic properties of the robot is defined. Furthermore, controller parameters to be utilized by the whole body controller are bound to a set of transport layers which enables the controller parameters to be accessed by external processes. The whole body controller is instantiated utilizing the compound task, constraint set, robot model and parameter bindings. In this manner, whole body controllers are effectively integrated with robots and their applications.



LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, **Published:**

SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ,

GW, KM, ML, MR, NE, SN, TD, TG).

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

INTEGRATION OF WHOLE BODY CONTROLLERS WITH ROBOTS

CROSS REFERENCE TO RELATED APPLICATIONS

[0001] This application claims priority to U.S. Provisional Patent Application Serial No. 62/166,927, "Middleware for Whole-Body Operational Space Control," filed May 27, 2015, which is incorporated by reference herein in its entirety.

GOVERNMENT INTERESTS

[0002] This invention was made with government support under Grant No. NNX12AQ99G awarded by the National Aeronautics and Space Administration. The U.S. government has certain rights in the invention

TECHNICAL FIELD

[0003] The present invention relates generally to humanoid robotics, and more particularly to integrating whole body controllers with modern state-of-the-art robots.

BACKGROUND

[0004] With growing research interest in humanoid robotics, robots have become increasingly proficient in performing many different, non-trivial tasks, such as running, jumping, climbing stairs and manipulating objects. In most cases, however, each of these tasks is addressed individually, and this imposes a fundamental limitation on the use of robots in the real world. While humans may occasionally be outperformed by robots in a single task, they are vastly more capable of adapting and combining behaviours to solve multiple different tasks. This flexibility allows humans to generalize their knowledge, and to successfully perform tasks that they have never explicitly faced before. This also opens the door for simultaneous execution of multiple tasks. To address these constraints, Whole Body Control (WBC) systems have been proposed as a promising research direction. They represent a wide range of complex movement skills in the form of low-dimensional task descriptors which are projected on to the robot's actuators thereby exploiting the full capabilities of the entire body.

[0005] WBC takes a holistic view of a multi-branched highly redundant robot like humanoids to achieve general coordinated behaviors. One of the first WBC algorithms is Whole Body

Operational Space Control (WBOSC), which provides the theoretical foundations for achieving operational space inverse dynamics, task prioritization, free floating degrees of freedom, contact constraints and internal forces. There is now a growing community of researchers in this field as exemplified by the recent formation of an IEEE technical committee on WBC. While the foundational theory and algorithms behind WBC have recently made great strides, less progress exists in software support, limiting the use of WBC today.

SUMMARY

[0006] In one embodiment of the present invention, a method for utilizing a mobile device as a motion-based controller comprises determining a distance between two or more speakers of a device to be controlled by the mobile device. The method further comprises receiving inaudible acoustic signals by the mobile device with a microphone from the device. The method additionally comprises recording the inaudible acoustic signals. Furthermore, the method comprises estimating a frequency shift using the recorded inaudible acoustic signals. Additionally, the method comprises estimating a velocity of the mobile device using the estimated frequency shift. In addition, the method comprises estimating distances the mobile device is located from each of the two or more speakers using the estimated velocity and a previous position of the mobile device. The method further comprises determining, by a processor, a current location of the mobile device using the estimated distances the mobile device is located from the two or more speakers, the distance between the two or more speakers of the device and the previous position of the mobile device.

[0007] Other forms of the embodiment of the method described above are in a system and in a computer program product.

[0008] The foregoing has outlined rather generally the features and technical advantages of one or more embodiments of the present invention in order that the detailed description of the present invention that follows may be better understood. Additional features and advantages of the present invention will be described hereinafter which may form the subject of the claims of the present invention.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] A better understanding of the present invention can be obtained when the following detailed description is considered in conjunction with the following drawings, in which:

[0010] Figure 1 illustrates a system for controlling a humanoid robot in accordance with an embodiment of the present invention;

[0011] Figure 2 illustrates an embodiment of the present invention of a hardware configuration of a computing system for controlling the humanoid robot;

[0012] Figure 3 is a flowchart of a method for controlling movement of the humanoid robot in accordance with an embodiment of the present invention;

[0013] Figure 4 illustrates the software abstractions that enable ControlIt! to instantiate and integrate general WBOSC controllers in accordance with an embodiment of the present invention;

[0014] Figure 5 is a Unified Modeling Language (UML) diagram that specifies the APIs of ControlIt!'s configuration software abstractions in accordance with an embodiment of the present invention;

[0015] Figure 6 is a UML class diagram that shows the tasks in ControlIt!'s task library and Proportional-Integral-Derivative (PID) controller that they use in accordance with an embodiment of the present invention;

[0016] Figure 7 is a UML class diagram that shows the constraints in ControlIt!'s constraint library in accordance with an embodiment of the present invention;

[0017] Figure 8 shows the class diagrams for the whole body control software abstractions in accordance with an embodiment of the present invention;

[0018] Figure 9 illustrates the two plugins in the WBC plugin library of ControlIt! in accordance with an embodiment of the present invention;

[0019] Figure 10 illustrates the hardware abstraction layer consisting of two abstract classes, the RobotInterface and the ServoClock, in accordance with an embodiment of the present invention;

[0020] Figure 11 illustrates the libraries of the robot interface and servo clock plugins in accordance with an embodiment of the present invention;

[0021] Figure 12A illustrates the API and class hierarchy of parameter reflection in accordance with an embodiment of the present invention;

[0022] Figure 12B illustrates the API and class hierarchy of parameter binding in accordance with an embodiment of the present invention;

[0023] Figure 12C illustrates the API of events in accordance with an embodiment of the present invention;

[0024] Figure 13A illustrates the servo thread that executes the actual servo loop in accordance with an embodiment of the present invention;

[0025] Figure 13B illustrates one of the child threads, the “ModelUpdater,” that is responsible for updating the robot model in accordance with an embodiment of the present invention;

[0026] Figure 13C illustrates the other child thread, the “TaskUpdater,” that is responsible for updating the states in each task in the compound task in accordance with an embodiment of the present invention; and

[0027] Figure 13D illustrates the operations of the CheckForUpdates state in accordance with an embodiment of the present invention.

DETAILED DESCRIPTION

[0028] In the following description, numerous specific details are set forth to provide a thorough understanding of the present invention. However, it will be apparent to those skilled in the art that the present invention may be practiced without such specific details. In other instances, well-known circuits have been shown in block diagram form in order not to obscure the present invention in unnecessary detail. For the most part, details considering timing considerations and the like have been omitted inasmuch as such details are not necessary to obtain a complete understanding of the present invention and are within the skills of persons of ordinary skill in the relevant art.

[0029] Referring now to the Figures in detail, Figure 1 illustrates a system 100 for controlling a humanoid robot in accordance with an embodiment of the present invention. Referring to Figure 1, system 100 includes a computing system 101 configured to control a robot 102 via a network 103. Computing system 101 may be a portable computing unit, a Personal Digital Assistant (PDA), a smartphone, a laptop computer, a mobile phone, a navigation device, a game console, a desktop computer system, a workstation, an Internet appliance and the like. A description of the hardware configuration of computing system 101 is provided below in connection with Figure 2.

[0030] Robot 102 may be any humanoid robot, such as a floating-base highly-redundant robot. As discussed above, robot 102 may be controlled by computing system 101 via network 103. Network 103 may be, for example, a local area network, a wide area network, a wireless wide area network, a circuit-switched telephone network, a Global System for Mobile Communications (GSM) network, Wireless Application Protocol (WAP) network, a WiFi network, an IEEE 802.11 standards network, various combinations thereof, etc. Other networks, whose descriptions are omitted here for brevity, may also be used in conjunction with system 100 of Figure 1 without departing from the scope of the present invention.

[0031] Referring now to Figure 2, Figure 2 illustrates an embodiment of the present invention of a hardware configuration of computing system 101 (Figure 1) for controlling robot 102. Computing system 101 has a processor 201 coupled to various other components by system bus 202. An operating system 203 runs on processor 201 and provides control and coordinates the functions of the various components of Figure 2. An application 204 in accordance with the principles of the present invention runs in conjunction with operating system 203 and provides

calls to operating system 203 where the calls implement the various functions or services to be performed by application 204. Application 204 may include, for example, a program for controlling the functionality of robot 102 as discussed herein.

[0032] Referring again to Figure 2, read-only memory ("ROM") 205 is coupled to system bus 202 and includes a basic input/output system ("BIOS") that controls certain basic functions of computing system 101. Random access memory ("RAM") 206 and disk adapter 207 are also coupled to system bus 202. It should be noted that software components including operating system 203 and application 204 may be loaded into RAM 206, which may be computing system's 101 main memory for execution. Disk adapter 207 may be an integrated drive electronics ("IDE") adapter that communicates with a disk unit 208, e.g., disk drive. It is noted that the program for controlling the functionality of robot 102 as discussed herein may reside in disk unit 208 or in application 204.

[0033] Computing system 101 may further include a communications adapter 209 coupled to bus 202. Communications adapter 209 interconnects bus 202 with an outside network (e.g., network 103) thereby enabling computing system 101 to communicate with robot 102.

[0034] I/O devices may also be connected to computing system 101 via a user interface adapter 210 and a display adapter 211. Keyboard 212, mouse 213 and speaker 214 may all be interconnected to bus 202 through user interface adapter 210. A display monitor 215 may be connected to system bus 202 by display adapter 211. In this manner, a user is capable of inputting to computing system 101 through keyboard 212 or mouse 213 and receiving output from computing system 101 via display 215 or speaker 214.

[0035] The present invention may be a system, a method, and/or a computer program product. The computer program product may include a computer readable storage medium (or media) having computer readable program instructions thereon for causing a processor to carry out aspects of the present invention.

[0036] The computer readable storage medium can be a tangible device that can retain and store instructions for use by an instruction execution device. The computer readable storage medium may be, for example, but is not limited to, an electronic storage device, a magnetic storage device, an optical storage device, an electromagnetic storage device, a semiconductor storage device, or any suitable combination of the foregoing. A non-exhaustive list of more specific

examples of the computer readable storage medium includes the following: a portable computer diskette, a hard disk, a random access memory (RAM), a read-only memory (ROM), an erasable programmable read-only memory (EPROM or Flash memory), a static random access memory (SRAM), a portable compact disc read-only memory (CD-ROM), a digital versatile disk (DVD), a memory stick, a floppy disk, a mechanically encoded device such as punch-cards or raised structures in a groove having instructions recorded thereon, and any suitable combination of the foregoing. A computer readable storage medium, as used herein, is not to be construed as being transitory signals *per se*, such as radio waves or other freely propagating electromagnetic waves, electromagnetic waves propagating through a waveguide or other transmission media (e.g., light pulses passing through a fiber-optic cable), or electrical signals transmitted through a wire.

[0037] Computer readable program instructions described herein can be downloaded to respective computing/processing devices from a computer readable storage medium or to an external computer or external storage device via a network, for example, the Internet, a local area network, a wide area network and/or a wireless network. The network may comprise copper transmission cables, optical transmission fibers, wireless transmission, routers, firewalls, switches, gateway computers and/or edge servers. A network adapter card or network interface in each computing/processing device receives computer readable program instructions from the network and forwards the computer readable program instructions for storage in a computer readable storage medium within the respective computing/processing device.

[0038] Computer readable program instructions for carrying out operations of the present invention may be assembler instructions, instruction-set-architecture (ISA) instructions, machine instructions, machine dependent instructions, microcode, firmware instructions, state-setting data, or either source code or object code written in any combination of one or more programming languages, including an object oriented programming language such as Smalltalk, C++ or the like, and conventional procedural programming languages, such as the "C" programming language or similar programming languages. The computer readable program instructions may execute entirely on the user's computer, partly on the user's computer, as a stand-alone software package, partly on the user's computer and partly on a remote computer or entirely on the remote computer or server. In the latter scenario, the remote computer may be connected to the user's computer through any type of network, including a local area network (LAN) or a wide area network (WAN), or the connection may be made to an external computer

(for example, through the Internet using an Internet Service Provider). In some embodiments, electronic circuitry including, for example, programmable logic circuitry, field-programmable gate arrays (FPGA), or programmable logic arrays (PLA) may execute the computer readable program instructions by utilizing state information of the computer readable program instructions to personalize the electronic circuitry, in order to perform aspects of the present invention.

[0039] Aspects of the present invention are described herein with reference to flowchart illustrations and/or block diagrams of methods, apparatus (systems), and computer program products according to embodiments of the invention. It will be understood that each block of the flowchart illustrations and/or block diagrams, and combinations of blocks in the flowchart illustrations and/or block diagrams, can be implemented by computer readable program instructions.

[0040] These computer readable program instructions may be provided to a processor of a general purpose computer, special purpose computer, or other programmable data processing apparatus to produce a machine, such that the instructions, which execute via the processor of the computer or other programmable data processing apparatus, create means for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks. These computer readable program instructions may also be stored in a computer readable storage medium that can direct a computer, a programmable data processing apparatus, and/or other devices to function in a particular manner, such that the computer readable storage medium having instructions stored therein comprises an article of manufacture including instructions which implement aspects of the function/act specified in the flowchart and/or block diagram block or blocks.

[0041] The computer readable program instructions may also be loaded onto a computer, other programmable data processing apparatus, or other device to cause a series of operational steps to be performed on the computer, other programmable apparatus or other device to produce a computer implemented process, such that the instructions which execute on the computer, other programmable apparatus, or other device implement the functions/acts specified in the flowchart and/or block diagram block or blocks.

[0042] The flowchart and block diagrams in the Figures illustrate the architecture, functionality, and operation of possible implementations of systems, methods, and computer program products

according to various embodiments of the present invention. In this regard, each block in the flowchart or block diagrams may represent a module, segment, or portion of instructions, which comprises one or more executable instructions for implementing the specified logical function(s). In some alternative implementations, the functions noted in the block may occur out of the order noted in the figures. For example, two blocks shown in succession may, in fact, be executed substantially concurrently, or the blocks may sometimes be executed in the reverse order, depending upon the functionality involved. It will also be noted that each block of the block diagrams and/or flowchart illustration, and combinations of blocks in the block diagrams and/or flowchart illustration, can be implemented by special purpose hardware-based systems that perform the specified functions or acts or carry out combinations of special purpose hardware and computer instructions.

[0043] As discussed in the Background section, WBC takes a holistic view of a multi-branched highly redundant robot like humanoids to achieve general coordinated behaviors. One of the first WBC algorithms is Whole Body Operational Space Control (WBOSC), which provides the theoretical foundations for achieving operational space inverse dynamics, task prioritization, free floating degrees of freedom, contact constraints and internal forces. There is now a growing community of researchers in this field as exemplified by the recent formation of an IEEE technical committee on WBC. While the foundational theory and algorithms behind WBC have recently made great strides, less progress exists in software support, limiting the use of WBC today.

[0044] The principles of the present invention provide a software middleware package that enables the integration of whole body control controllers with the robots and their applications as discussed herein.

[0045] As discussed below, a middleware, referred to herein as ControlIt!, enables WBOSC controllers to be instantiated and is designed for systems integration, extensibility, high performance and use by both WBC researchers and the general public. Instantiating a WBOSC controller consists of defining a prioritized compound task that defines the operational space objectives and lower priority goal postures that the controller should achieve, and a constraint set that specifies the natural physical constraints of robot 102. Systems integration is achieved through a parameter binding mechanism that enables external processes to access WBOSC

parameters through various transport layers, and a set of introspection tools for gaining insight into the controller's state at runtime. ControlIt! is extensible through the use of plugins that enable the addition of new WBC primitives and support for new robot platforms. High performance is achieved by using state-of-the-art software libraries and multiple-threads that enable ControlIt! to offer higher servo frequencies relative to previous WBOSC implementations.

[0046] A discussion regarding a method for ControlIt! to control robot 102 is provided below in connection with Figure 3.

[0047] Figure 3 is a flowchart of a method 300 for controlling movement of a humanoid robot 102 in accordance with an embodiment of the present invention.

[0048] Referring to Figure 3, in conjunction with Figures 1-2, in step 301, computing system 101 defines a compound task that defines operational space objectives and lower priority goal postures that a whole body controller should achieve.

[0049] In step 302, computing system 101 defines a constraint set that specifies the natural physical constraints of robot 102.

[0050] In step 303, computing system 101 defines a robot model that computes the kinematic and dynamic properties of robot 102.

[0051] In step 304, computing system 101 defines a hardware abstraction layer including a robot interface and a servo clock. In one embodiment, the robot interface is responsible for obtaining a joint state of robot 102 and sending a command from the whole body controller to robot 102. In one embodiment, the servo clock is responsible for initializing the whole body controller.

[0052] In step 305, computing system 101 defines controller parameters to be utilized by the whole body controller.

[0053] In step 306, computing system 101 binds the controller parameters to a set of transport layers which enables the controller parameters to be accessed by external processes.

[0054] In step 307, computing system 101 stores the definition of events which contains a logical expression over the controller parameters thereby enabling parameter changes to trigger execution of the external processes without the use of polling.

[0055] In step 308, computing system 101 utilizes a set of service-based introspection capabilities to enable the external processes to query controller properties as it is running.

[0056] In step 309, computing system 101 instantiates the whole body controller utilizing the compound task, constraint set, robot model, hardware abstraction layer, parameter bindings and events.

[0057] A more detailed description of each of these steps is provided below.

[0058] A discussion regarding ControlIt!'s software architecture is now deemed appropriate. In one embodiment, many of these components either instantiate plugins or are implemented by plugins. The use of plugins enables ControlIt! to be extensible in terms of supporting different robots and applications. A discussion regarding the mechanisms for configuring and integrating ControlIt! into a larger system is also discussed further below. This includes the parameter reflection, binding, and event signaling mechanisms, and YAML (human-readable data serialization language) specification files. Finally, a description of ControlIt!'s multi-threaded architecture is provided further below.

[0059] Referring to Figure 4, Figure 4 illustrates the software abstractions that enable ControlIt! to instantiate and integrate general WBOSC controllers in accordance with an embodiment of the present invention. The abstractions that are extendable via dynamically loadable plugins are tasks 401, constraints 402, the whole body controller 403, the servo clock 404 (identified as "ServoClock" in Figure 4) and the robot interface 405 (identified as "RobotInterface" in Figure 4). Non-extensible components include the compound task 406 (identified as "CompoundTask" in Figure 4), robot model 407 (identified as "RobotModel" in Figure 4), constraint set 408 (identified as "ConstraintSet" in Figure 4) and coordinator 409. Coordinator 409 implements the servo loop and uses all of the other abstractions except for servo clock 404, which implements the servo thread and controls when coordinator 409 executes the next cycle of the servo loop. The software abstractions can be divided into three general categories: configuration 410, whole body control 411, and hardware abstraction 412.

[0060] In one embodiment, configuration software abstractions include robot model 407, compound task 406, and constraint set 408. Their APIs and attributes are shown in Figure 5. Figure 5 is a Unified Modeling Language (UML) diagram that specifies the APIs of ControlIt!'s

configuration software abstractions in accordance with an embodiment of the present invention. They are used to specify the objectives and constraints of the whole body controller.

[0061] Referring to Figure 5, in conjunction with Figure 4, robot model 407 determines the kinematic and dynamic properties of robot 102 and builds upon the model provided by the Rigid Body Dynamics Library (RBDL), which includes algorithms for computing forward and inverse kinematics and dynamics and frame transformations. The kinematic and dynamic values provided by the model are only estimates and may be incorrect, necessitating the use of a whole body feedback controller. The API of robot model 407 includes methods for saving and obtaining the joint state 501 (identified as “JointState” in Figure 5) and getting properties of robot 102 like the joint space inertia matrix and gravity compensation vector. There are also methods for obtaining the joint order within whole body controller 403. A reference to the constraint set is kept within robot model 407 to determine which joints are virtual, real and actuated. Furthermore, in one embodiment, the API of robot model 407 includes methods for saving and obtaining the task state 502 (identified as “TaskState” in Figure 5).

[0062] Compound task 406 and constraint set 408 contain lists of tasks and constraints, respectively. Tasks and constraints are abstract; concrete implementations are added to ControlIt! through plugins. Both have names and types for easy identification and can be enabled or disabled based on context. Task 401 represents an operational or postural objective for whole body controller 403 to achieve. Concrete task implementations contain goal parameters that, in combination with the robot model, produce an error. The error is used by controller 403 inside task 401 to generate a task-space effort command, which is accessible through the getCommand() method and may be in units of force or torque. In addition to the command, task 401 also provides a Jacobian that maps from task space to joint space. Compound task 406 combines the commands and Jacobians of the enabled tasks 401 and relays this information to whole body controller 403. Specifically, for each priority level, compound task 406 vertically concatenates the Jacobians and commands belonging to tasks 401 at the priority level. The WBOSC algorithm uses these concatenated Jacobian and command matrices to support task prioritization and multiple tasks at the same priority level.

[0063] To encourage code reuse and enable support for basic applications, ControlIt! comes with a task library containing commonly used tasks. The tasks within this library are shown in

Figure 6. Figure 6 is a UML class diagram that shows the tasks in ControlIt!'s task library and Proportional-Integral-Derivative (PID) controller 601 (identified as "PIDController" in Figure 6) that they use in accordance with an embodiment of the present invention. In one embodiment, combinations of these tasks 401 specify the operational space and postural objectives of the whole body controller and collectively form compound task 406. Concrete tasks are implemented as dynamically loadable plugins. ControlIt! can be easily extended with new tasks via the plugin mechanism.

[0064] Referring to Figure 6, in conjunction with Figures 4-5, Figure 6 illustrates six tasks 401 in the library: joint position 602 (identified as "JointPositionTask" in Figure 6), 2D/3D orientation 603, 604, respectively (identified as "2DOrientationTask" and "3DOrientationTask," respectively, in Figure 6), center of mass (COM) 605 (identified as "COMTask" in Figure 6), cartesian position 606 (identified as "CartesianPositionTask 606 in Figure 6) and center of pressure (COP) 607 (identified as "COPTask" in Figure 6). While Figure 6 illustrates six tasks 401, the principles of the present invention are not to be limited to such a specific number of tasks 401. More tasks can be added to the library by introducing additional plugins. For example, an external force task may be added that controls robot 102 to assert a certain amount of force against an external obstacle. In addition, an internal force task may be added to control the internal tensions between multiple contact points.

[0065] In one embodiment, all of the tasks 602-607 make use of PID controller 601. This feedback controller generates the task-space command based on the current error and gains.

[0066] Joint position task 602 directly specifies the goal positions, velocities and accelerations of every joint in robot 102. It typically defines the desired "posture" of robot 102, which is not an operational space objective but accounts for situations where there is sufficient redundancy within robot 102 to result in non-deterministic behavior when no posture is defined. Specifically, a posture task is necessary when the null space of all higher priority tasks and constraints is not nil, and the best practice is to include one as the lowest priority task in compound task 406. Joint position task 602 has an input parameter called goalAcceleration to enable smooth transitions between joint positions. The goal acceleration is a desired acceleration that is added as a "feedforward" command to the control law. In one embodiment, the

currentAcceleration output parameter is a copy of the goalAcceleration parameter and is used for debugging purposes.

[0067] 2D and 3D orientation tasks 603, 604 are used to control the orientation of a link on robot 102. They differ in terms of how the orientations are specified. In one embodiment, the 2D orientation is specified by a vector in the frame of the body being oriented, whereas, the 3D orientation is specified using a quaternion. The purpose of providing a 2D orientation task 603 even though a 3D orientation could be used is to reduce computational overhead when only two degrees of orientation control is required. In one embodiment, the 2D orientation task 603 does not include a goalAngularVelocity input parameter because its current implementation assumes the goal velocity is always zero. This assumption can be easily removed by modifying the control law to include a non-zero goal velocity.

[0068] Center Of Mass (COM) task 605 controls the location of the robot's COM, which is derived from robot model 407. It is useful when balancing since it can ensure that the robot's configuration always results in the COM being above the convex polygon surrounding the supports holding robot 102 up.

[0069] Center Of Pressure (COP) task 607 controls the center of pressure of a link that is in contact with the ground. It is particularly useful for biped robots 102 containing feet since it can help ensure that the COP of a foot remains within the boundaries of the foot thereby preventing the foot from rolling.

[0070] Cartesian position task 606 controls the operational space location of a point on robot 102. Typically, this means the location of an end effector in a frame that is specified by the user and is by default the world frame.

[0071] With respect to the constraints, a constraint specifies natural physical limits of robot 102. In one embodiment, there are two types of constraints 402: contact constraints and transmission constraints. Contact constraints specify places where a robot 102 touches the environment. Transmission constraints specify dependences between joints, which occur when, for example, joints are co-actuated. The parent constraint class includes methods for obtaining the number of Degrees of Freedom (DOFs) that are constrained and the Jacobian of the constraint. Contact constraints have a getJoint() method that specifies the parent joint of the link that is constrained. Transmission constraints have a master joint that is actuated and a set of slave joints that are co-

actuated with the master joint. Unlike tasks 401, constraints 402 do not have commands since they simply specify the nullspace within which all tasks 401 must operate. Like compound task 406, constraint set 408 computes a Jacobian that is the vertical concatenation of all the constraint Jacobians. In addition, it provides an update method that computes accessors for these matrices and methods for determining whether a particular joint is constrained. Whole body controller 403 uses this information to ensure all of the constraints are met. While it is true that the contact constraints are mathematically similar to tasks 401 without an error term, it may be desirable to distinguish between the two since they serve significantly different purposes: tasks 401 denote a user's control objectives while constraints 402 denote the physical limits of robot 102. By separating tasks 401 and constraints 402, the API will be easier to extend to support optimization based controllers with inequality constraints.

[0072] Referring now to Figure 7, Figure 7 is a UML class diagram that shows the constraints 402 in ControlIt!'s constraint library in accordance with an embodiment of the present invention. Combinations of these constraints specify natural physical limits of robot 102 and constitute constraint set 408. Concrete constraints are implemented as dynamically loadable plugins. Additional constraints can be easily added via the plugin mechanism.

[0073] Referring to Figure 7, in conjunction with Figures 4-6, contact constraints 701 (see Figures 5 and 7) (identified as "ContactConstraint" in Figures 5 and 7) include the flat contact constraint 702 (identified as "FlatContactConstraint" in Figure 7), omni wheel contact constraint 703 (identified as "OmniWheelContactConstraint" in Figure 7) and point contact constraint 704 (identified "PointContactConstraint" in Figure 7). The contact constraint 701 restricts both link translation and rotation. Omni wheel contact constraint 703 restricts one rotational DOF and one translational DOF based on the current orientation of the wheel. Point contact constraint 704 restricts just link translation. One transmission constraint 705 (see Figures 5 and 7) (identified as "TransmissionConstraint" in Figures 5 and 7) called coactuation constraint 706 (identified as "CoactuationConstraint" in Figure 7) is provided that enables ControlIt! to handle robots 102 with two co-actuated joints. It includes a transmission ratio specification to handle situations where the relationship between the master joint and slave joint is not one-to-one. Like the task library, the constraint library can easily be extended with new constraints via the plugin mechanism used by ControlIt!.

[0074] Figure 8 shows the class diagrams for the whole body control software abstractions in accordance with an embodiment of the present invention. Referring to Figure 8, in conjunction with Figure 4, in one embodiment, there are two classes: WBC 801 and Command 802. WBC 801 is an interface that contains a single computeCommand() method. This method takes as input robot model 407, which includes constraint set 408 and compound task 406. It performs the WBC computations that generate a command for each joint under its control and returns it within a command object. The command object specifies the desired position, velocity, effort and position controller gains. It is noted that not all of these variables need to be used. For example, a robot 102 that is purely effort controlled will only use the effort command. The optional fields within the command are included to support robots 102 with joints that are position or impedance controlled.

[0075] In one embodiment, whole body controller 403 within ControlIt! is dynamically loaded as a plugin using ROS pluginlib. Two plugins are currently available as shown in Figure 9. Figure 9 illustrates the two plugins in the WBC plugin library 900 of ControlIt! in accordance with an embodiment of the present invention. Referring to Figure 9, in conjunction with Figure 4, they include WBOSC and WBOSC Impedance 901, 902, respectively. The WBOSC plugin 901 implements the WBOSC algorithm. It computes the nullspace of constraint set 408 and projects the task commands through this nullspace. Task commands are iteratively included into the final command based on priority. The commands of tasks 401 at a particular priority level are projected through the nullspaces of all higher priority tasks 401 and constraint set 408. This ensures that all constraints 402 are met and that higher priority tasks 401 override lower priority tasks 401. The output of WBOSC 901 is an effort command that can be sent to effort controlled robots. The member variables within the WBOSC plugin 901 ensure that memory is pre-allocated, which reduces execution time jitter and thus increases real-time predictability.

[0076] To support impedance-controlled robots, ControlIt! also comes with the WBOSC impedance plugin 902. Unlike effort-controlled robots, impedance-controlled robots take more than just effort commands. Specifically, in addition to effort, impedance controllers also take desired position and velocity commands, and optionally position controller gains when controller gain scheduling is desired. The benefit of using impedance control is the ability to attain higher levels of impedance. This is possible since the position and velocity control loop can be closed by the embedded joint controller, which typically has a higher servo frequency and lower

communication latency than WBC controller 403. The WBOSC impedance plugin 902 extends the WBOSC plugin 901 with an internal model that converts the effort commands generated by the WBOSC algorithm into expected joint positions and velocities. The member variables within the WBOSC impedance plugin 902 that start with "qi_" hold the internal model's joint states. The prevUpdateTime member variable records when this internal model was last updated. Each time computeCommand is called, WBOSC impedance 902 computes the desired effort command using WBOSC 901. It then uses this effort command along with the robot model to determine the desired accelerations of each joint. WBOSC impedance 902 then updates the internal model based on these acceleration values, the time since the last update, the previous state of the internal model, and the actual position and velocity of the joints. The derived joint positions, velocities, and efforts are saved within a command object, which is returned.

[0077] To enable support for a wide variety of robot platforms, ControlIt! includes a hardware abstraction layer 412 consisting of two abstract classes, the robot interface 405 and the servo clock 402, as shown in Figure 10 in accordance with an embodiment of the present invention. Referring to Figure 10, in conjunction with Figure 4, concrete implementations are provided through dynamically loadable plugins. Robot interface 405 is responsible for obtaining the robot's joint state in robot state class 1001 and sending the command from whole body controller 403 to robot 102. For diagnostic purposes, it also publishes the state and command information onto Robot Operating System (ROS) topics using a real-time ROS topic publisher, which uses a thread-pool to offload the publishing process from the servo thread. Servo clock 402 instantiates the servo thread and contains a reference to a controller 1002, which is implemented by coordinator 409. Servo clock 402 is responsible for initializing controller 1002 by calling servoInit() and then periodically executing the servo loop by calling the servoUpdate() method. Initialization using the actual servo thread is needed to handle situations where certain initialization tasks 401 can only be done by the servo thread. This occurs, for example, when the servo thread is part of a real-time context meaning only it can initialize certain real-time resources.

[0078] ControlIt! includes libraries of robot interface and the servo clock plugins 405, 404, respectively, as shown in Figure 11 in accordance with an embodiment of the present invention. Referring to Figure 11, in conjunction with Figures 4 and 6, robot interface plugin 405 includes general ones that communicate with robot 102 via three different transport layers: ROS topics

(Robot Interface ROS Topic) 1101 (identified as “RobotInterfaceROSTopic” in Figure 11), UDP datagrams (Robot Interface UDP) 1102 (identified as “RobotInterfaceUDP” in Figure 11), and shared memory (Robot Interface SM) 1103 (identified as “RobotInterfaceSM” in Figure 11). These are meant for general use - ControlIt! includes generic Gazebo plugins and abstract classes that facilitate the creation of software adapters for allowing simulated and real robots to communicate with ControlIt! using these three transport layers. Among the three transport layers, shared memory 1103 has the lowest latency and is most reliable in terms of message loss. It uses the ROS shared memory interface package, which is based on boost's interprocess communication library.

[0079] In addition to general Robot Interface plugins, ControlIt! also includes two robot-specific plugins, one for “Dreamer” (Robot Interface Dreamer) 1104 (identified as “RobotInterfaceDreamer” in Figure 11), and one for “Valkyrie” (Robot Interface Valkyrie) 1105 (identified as “RobotInterfaceValkyrie” in Figure 11). The humanoid robot “Dreamer” was developed by the University of Texas at Austin and Meka Robotics. The humanoid robot “Valkyrie” was developed by National Aeronautics and Space Administration (NASA).

[0080] In one embodiment, Robot Interface Dreamer 1104 interfaces with a RealTime Application Interface (RTAI) real-time shared memory segment that is created by the robot's software interface called the M3 Server. It also implements separate PID controllers 601 for robot joints that are not controlled by WBC. They include the finger joints in the right hand, the left gripper joint, the neck joints and the head joints. In one embodiment, these joints are fixed from WBC's perspective.

[0081] Robot Interface Valkyrie 1105 interfaces with the shared memory segment created by Valkyrie's software interface. This involves integration with a controller manager provided by `ros_control` to gain access to robot resources.

[0082] ControlIt! includes several servo clock plugins 402 to enable flexibility in the way the servo thread is instantiated and configured to be periodic. In one embodiment, the servo clock plugin library includes plugins for supporting servo threads based on a ROS timer 1106 (identified as “ServoClockROS” in Figure 11), a C++ `std::chrono` timer 1107 (identified as “ServoClockChrono” in Figure 11), or an RTAI timer 1108 (identified as “ServoClockRTAI” in Figure 11).

[0083] Support for configuration and integration is important because as a middleware ControlIt! is expected to be (1) used in many different applications and hardware platforms that require different whole body controllers and (2) just one component in a complex application consisting of many components. In addition, ControlIt!'s configuration and integration capabilities directly impacts the middleware's usability, which needs to be high to achieve the goal of widespread use. ControlIt! supports integration through four mechanisms: (1) parameter reflection, which exposes controller parameters to other objects within ControlIt! and is used by the other two mechanisms, (2) parameter binding, which enables the parameters to be connected to external processes through an extensible set of transport layers, (3) events, which enable parameter changes to trigger the execution of external processes without the use of polling, and (4) services, which enable external processes to query information about the controller. ControlIt! supports configuration through scripts that enable users to specify the structure of compound task 406 and constraint set 408, the type of whole body controller 403 and hardware interface to use, the initial values of the parameters, the parameter bindings, and the events. These scripts are interpreted during ControlIt!'s initialization to automatically instantiate the desired whole body controller 403 and integrate it into the rest of the system. Details of ControlIt!'s support for configuration and integration are now discussed.

[0084] Parameter reflection defines a Parameter Reflection parent class through which child class member variables can be exposed to other objects within ControlIt!. The API and class hierarchy of the Parameter Reflection class is shown in Figure 12A in accordance with an embodiment of the present invention. Parameter reflection enables internal control parameters to be exposed to other objects within ControlIt!. It consists of an abstract parent called Parameter Reflection 1201 (identified as "ParameterReflection" in Figure 12A) that provides methods for declaring and looking up parameters. When a parameter is declared, it is encapsulated within a parameter object 1202, which contains a name, pointer to the actual variable, a list of bindings and a method to set the parameter's value. Subclasses of Parameter Reflection 1201 are able to declare their member variables as parameters and thus make them compatible with ControlIt!'s parameter binding and event mechanism, such as the Reflection Registry 1203 (identified as "ReflectionRegistry" in Figure 12A).

[0085] Parameter binding enables the integration of ControlIt! with other processes in the system by connecting parameters to an extensible set of transport layers. Its API and class

hierarchy are shown in Figure 12B in accordance with an embodiment of the present invention. The classes that constitute the parameter binding mechanism consist of a Binding Manager 1204 (identified as “BindingManager” in Figure 12B) that maintains a set of Binding Factory objects 1205 (identified as “BindingFactory” in Figure 12B) (e.g., Binding Factory ROS 1206 and Binding Factory SM 1207 which are identified as “BindingFactoryROS” and “BindingFactorySM,” respectively in Figure 12B) that actually create the bindings, and a Binding Config object 1208 (identified as “BindingConfig” in Figure 12B) that specifies properties of a binding. The required properties include the binding direction (either input or output), the transport type, which is a string that must match the name of a binding provided by a Binding Factory plugin 1205, and a topic to which the parameter is bound. Binding Config 1208 also contains an extensible list of name-value properties that is transport protocol specific. For example, transport-specific parameters for ROS topic output bindings include the publish rate, the queue size and whether the latest value published should be latched.

[0086] During the initialization process, Binding Config objects 1208 are stored as parameters within a Parameter Reflection object 1201, which is passed to Binding Manager 1204. Binding Manager 1204 searches through its Binding Factory objects 1205, which are dynamically loaded via plugins, for factories that are able to create the desired binding. In one embodiment, the bindings in ControlIt's binding library include input and output bindings 1209 for ROS topics (e.g., Input Binding ROS 1210, Output Binding ROS 1211 which are identified as “InputBindingROS” and “OutputBindingROS,” respectively, in Figure 12B) and shared memory topics (e.g., Input Binding SM 1212, Output Binding SM 1213 which are identified as “InputBindingSM” and “OutputBindingSM,” respectively, in Figure 12B). The newly created Binding objects are stored in the parameter's parameter object. When a parameter's value is set via Parameter.set(), the new value is transmitted through output bindings to which the parameter is bound. This enables changes in ControlIt! parameters to be published onto various transport layers and topics notifying external processes of the latest values of the parameters. Similarly, when an external process publishes a value onto a transport layer and topic to which a parameter is bound via an input binding, the parameter's value is updated to be the published value. This enables, for example, external processes to dynamically change a task's references or controller gains, which is necessary for integration.

[0087] Events 1214 contain a logical expression over parameters that are interpreted via muParser, an open-source math parser library. Its API is shown in Figure 12C in accordance with an embodiment of the present invention. In one embodiment, events 1214 are stored in the Parameter Reflection parent class. The servo thread calls ParameterReflection.emitEvents() at the end of every servo cycle. The names of events whose condition expression evaluates to true are published on ROS topic /[controller name]/events. Events 1214 contain a Boolean variable called "enabled" that is used to prevent an event from continuously firing when the condition expression remains true since this would likely flood the events ROS topic. Instead, events maintain a fire-once semantic meaning they only fire when the condition expression changes from false to true.

[0088] To further assist ControlIt! integration, into a larger system, ControlIt! may include a set of service-based introspection capabilities. Unlike ROS topics, which are asynchronous unidirectional, ROS services are bi-directional and synchronous. ControlIt! uses this capability to enable external processes to query certain controller properties as it is running. For example, two often-used services include /[controllername]/diagnostics/getTaskParameters, which returns a list of all tasks in compound task 406, the parameters, and their parameter values, and /[controller name]/diagnostics/getRealJointIndices, which returns the ordering of all real joints in robot 102. This is useful to determine the joint order when updating the reference positions of a posture task or interpreting the meaning of the posture task's error vector.

[0089] As previously mentioned, ControlIt! supports script-based configuration specification and initialization enabling integration into different applications and platforms without being recompiled. This is used given the plethora of properties that are defined and the wide range of anticipated applications and hardware platforms. To instantiate whole body controller 403 using ControlIt!, the user may specify many things including compound task 406, constraint set 408, whole body controller 403, robot interface 405, servo clock 404, initial parameter values, parameter bindings and events 1214. In addition, there are numerous controller parameters that can be specified, e.g., which groups of joints should be coupled, whether to enforce joint effort limits, whether to enforce joint position limits, whether to enforce joint velocity limits, specifying which joints should not be gravity compensated and controlling how much log information is generated during runtime. ControlIt! enables users to define the primary WBC configuration and integration abstractions including tasks, constraints, compound tasks,

constraint set, parameter bindings and events. In one embodiment, the remaining parameters are defined through the ROS parameter server.

[0090] In one embodiment, higher servo frequencies can be achieved by decreasing the amount of computation in the servo loop. The amount of computation can be reduced because robots typically move very little during one servo period, which is usually 1 ms. Thus, the state that depends on the robot configuration like the robot model 407 and task Jacobians often do not need to be updated every servo cycle. ControlIt! takes advantage of this possibility by offloading the updating of robot model 407 and the task states 502, which include the task Jacobians, into child threads. Specifically, in one embodiment, ControlIt! uses three threads as shown in Figures 13A-13C in accordance with an embodiment of the present invention. They include (1) a servo thread 1301 that executes the actual servo loop as shown in Figure 13A, (2) a ModelUpdater thread 1302 that is responsible for updating robot model 407, which includes the kinematics, inertia matrix, gravity compensation vector, constraint set 408 and the virtual linkage model as shown in Figure 13B, and (3) a TaskUpdater thread 1303 that is responsible for updating the states 502 of each task 401 in compound task 406, which includes the task Jacobians, as shown in Figure 13C. Referring to Figures 13A-13C, in conjunction with Figures 4-11 and 12A-12C, the servo thread is instantiated by the servo clock 404 and can thus be real-time when, for example, servo clock RTAI 1108 is used. ModelUpdater and TaskUpdater are child threads 1302, 1303, respectively, that do not operate in a real-time manner. From a high-level perspective, servo 1301 provides ModelUpdater 1302 with the latest joint states. ModelUpdater 1302 uses this information to update robot model 407 in parallel with servo thread 1301, and provides the updated robot model 407 to servo 1301 when complete. Whenever robot model 407 is updated, servo thread 1301 provides the updated model to TaskUpdater thread 1303, which updates task states 502. These updated task states 502 are then provided to servo thread 1301. Details on how this process is achieved in a manner that is non-blocking and safe are now discussed.

[0091] Two key requirements of the multi-threaded architecture are (1) servo thread 1301 is not blocked and (2) there should not be any race conditions between threads 1301-1303. The first requirement implies that the servo thread 1301 cannot call the blocking lock() method on the mutexes protecting the shared states between it and the child threads 1302, 1303. Instead, it can only call the non-blocking try lock() method, which returns immediately if the lock is not

obtainable. ControlIt!'s multi-threaded architecture is thus structured to only require calls to try lock() within servo thread 1301. To prevent race conditions between threads, two copies of the robot model 407 and task state 502 are maintained: an "active" copy that is used by servo thread 1301, and an "inactive" one that is updated by the non-servo threads 1302, 1303. Updates from the child threads 1302, 1303 are provided to servo thread 1301 by swapping the active and inactive states. This swapping is done by servo thread 1301 in a non-blocking and opportunistic manner.

[0092] Figures 13A and 13B show how servo thread 1301 passes the latest joint state information to ModelUpdater thread 1302 and trigger it to execute. After obtaining the latest joint states by calling RobotInterface.read() and checking for updates from the child threads 1302, 1303 by executing the CheckForUpdates FSM, servo thread 1301 attempts to obtain the lock on the mutex protecting the inactive robot model by calling ModelUpdater.tryLock(). If it is able to obtain the lock on the mutex, it saves the latest joint states in the inactive robot model 407 and then triggers ModelUpdater thread 1302 to execute by calling ModelUpdater.unlockAndUpdate(). As the name of this method implies, servo thread 1301 also releases the lock on the inactive model thereby allowing ModelUpdater thread 1302 to access and update the inactive robot model 407. If servo thread 1301 fails to obtain the lock on the inactive model, ModelUpdater thread 1302 is busy updating the inactive model. In this situation, servo thread 1301 continues without updating the inactive model.

[0093] To prevent race conditions between servo thread 1301 and the child threads 1302, 1303, updates from child threads 1302, 1303 are opportunistically pulled by servo thread 1301. This is because the child threads 1302, 1303 operate on inactive versions of robot model 407 and task states 502, and only servo thread 1301 can swap the active and inactive versions. There are two points in the servo loop where servo thread 1301 obtains updates from the child threads 1302, 1303. This is shown by the two "CheckForUpdates" states in the left side of Figure 13A. They occur immediately after obtaining the latest joint states by calling RobotInterface.read(), and immediately after triggering ModelUpdater thread 1302 to run or failing to obtain the lock on the inactive robot model 407. In one embodiment, more checks for updates could be interspersed throughout the servo loop.

[0094] The operations of the CheckForUpdates state are shown in Figure 13D in accordance with an embodiment of the present invention. Servo thread 1301 first obtains task state updates and then checks whether TaskUpdater thread 1303 is idle. If it is idle, servo thread 1301 again checks for updated task states. This is to account for the following degenerate thread interleaving during the previous check for updated task states that would result in the permanent loss of updated task state:

(1) Servo thread 1301 begins to check some of the tasks for updated states.

(2) TaskUpdater thread 1303 updates all of the tasks 401 including those that were just checked by servo thread 1301 and returns to idle state. It is noted that this is possible even if servo thread 1301 is real-time and has higher priority since TaskUpdater 1303 may be executing on a different CPU core.

(3) Servo thread 1301 completes checking the remainder of the tasks 401 for updates. In the above scenario, tasks 401 that were checked in step 1 would have updated states that would be lost without servo-thread 1301 re-checking for them after it confirms that TaskUpdater 1303 is idle. In a worst-case scenario, TaskUpdater thread 1303 may update all of the tasks 401 after servo thread 1301 checks for updates but before it checks whether TaskUpdater 1303 is idle, resulting in the loss of the updated state from every task 401. The loss of the updated task state is not acceptable despite the presence of future update rounds since it is theoretically possible for the updated states of the same tasks 401 to be continuously lost during every update round. While seemingly improbable, this "task update starvation" problem was actually observed and thus discovered while testing ControlIt! on Valkyrie.

[0095] After verifying that TaskUpdater thread 1303 is idle and ensuring all of the updated task states were obtained, servo thread 1301 next checks for an updated robot model 407 by calling ModelUpdater.checkUpdate(). This method switches to the updated robot model 407 if one is available. If model 407 was updated, servo thread 1301 then calls TaskUpdater.updateTasks() passing it the updated robot model 407. This method is non-blocking since TaskUpdater 1303 is idle. It triggers TaskUpdater 1303 to update the states 502 of each task 401 in compound task 406. It is noted that if robot model 407 was not updated, servo thread 1301 does not call TaskUpdater.updateTasks() since task state updates are based on changes in robot model 407.

[0096] With the increasing availability of sophisticated multi-branched highly-redundant robots targeted for general applications, whole body controllers will likely become an essential component in advanced human-centered robotics. As discussed herein, ControlIt! is middleware that defines a software architecture and set of APIs for instantiating and configuring whole body controllers, integrating them into larger systems and different robot platforms and enabling high performance via multi-threading. Experiments using the present invention have demonstrated high performance with servo computational latencies of about 0.5 ms, which is an order of magnitude faster than the 5 ms established by the previous state-of-the-art prototype implementation of WBOSC.

[0097] In summary, WBC is a rich and vibrant though fragmented research area today with numerous algorithms and implementations that are not cross-compatible and thus difficult to compare in hardware. ControlIt! is a unifying framework for supporting the development and study of whole body operational space controllers and their integration into useful advanced robotic applications.

[0098] The descriptions of the various embodiments of the present invention have been presented for purposes of illustration, but are not intended to be exhaustive or limited to the embodiments disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art without departing from the scope and spirit of the described embodiments. The terminology used herein was chosen to best explain the principles of the embodiments, the practical application or technical improvement over technologies found in the marketplace, or to enable others of ordinary skill in the art to understand the embodiments disclosed herein.

CLAIMS:

1. A method for controlling a humanoid robot, the method comprising:
defining a compound task that defines operational space objectives and lower priority goal postures that a whole body controller should achieve;
defining a constraint set that specifies natural physical constraints of a robot;
defining a robot model that computes kinematic and dynamic properties of said robot;
defining controller parameters to be utilized by said whole body controller;
binding said controller parameters to a set of transport layers which enables said controller parameters to be accessed by external processes; and
instantiating, by a processor, said whole body controller utilizing said compound task, said constraint set, said robot model and said parameter bindings.
2. The method as recited in claim 1 further comprising:
defining a hardware abstraction layer comprising a robot interface and a servo clock, wherein said robot interface is responsible for obtaining a joint state of said robot and sending a command from said whole body controller to said robot, wherein said servo clock is responsible for initializing said whole body controller.
3. The method as recited in claim 1 further comprising:
storing definition of events which contains a logical expression over said controller parameters thereby enabling parameter changes to trigger execution of external processes without the use of polling.
4. The method as recited in claim 1 further comprising:
utilizing a set of service-based introspection capabilities to enable external processes to query controller properties as it is running.
5. The method as recited in claim 1, wherein said whole body controller is instantiated utilizing said compound task, said constraint set, said robot model and said parameter bindings in a multi-threaded system.

6. The method as recited in claim 1, wherein said controller parameters specify one or more of the following: which groups of joints should be coupled, whether to enforce joint effort limits, whether to enforce joint position limits, whether to enforce joint velocity limits, specifying which joints should not be gravity compensated and controlling how much log information is generated during runtime.

7. The method as recited in claim 1, wherein said compound task comprises a plurality of tasks, wherein said plurality of tasks comprises one or more of the following: joint position, two-dimensional orientation, three-dimensional orientation, center of mass, Cartesian position and center of pressure.

8. The method as recited in claim 1, wherein said constraint set comprises a plurality of constraints, wherein said plurality of constraints comprises one or more of the following: contact constraints and transmission constraints.

9. A computer program product for controlling a humanoid robot, the computer program product comprising a computer readable storage medium having program code embodied therewith, the program code comprising the programming instructions for:

- defining a compound task that defines operational space objectives and lower priority goal postures that a whole body controller should achieve;

- defining a constraint set that specifies natural physical constraints of a robot;

- defining a robot model that computes kinematic and dynamic properties of said robot;

- defining controller parameters to be utilized by said whole body controller;

- binding said controller parameters to a set of transport layers which enables said controller parameters to be accessed by external processes; and

- instantiating said whole body controller utilizing said compound task, said constraint set, said robot model and said parameter bindings.

10. The computer program product as recited in claim 9, wherein the program code further comprises the programming instructions for:

- defining a hardware abstraction layer comprising a robot interface and a servo clock, wherein said robot interface is responsible for obtaining a joint state of said robot and sending a

command from said whole body controller to said robot, wherein said servo clock is responsible for initializing said whole body controller.

11. The computer program product as recited in claim 9, wherein the program code further comprises the programming instructions for:

storing definition of events which contains a logical expression over said controller parameters thereby enabling parameter changes to trigger execution of external processes without the use of polling.

12. The computer program product as recited in claim 9, wherein the program code further comprises the programming instructions for:

utilizing a set of service-based introspection capabilities to enable external processes to query controller properties as it is running.

13. The computer program product as recited in claim 9, wherein said whole body controller is instantiated utilizing said compound task, said constraint set, said robot model and said parameter bindings in a multi-threaded system.

14. The computer program product as recited in claim 9, wherein said controller parameters specify one or more of the following: which groups of joints should be coupled, whether to enforce joint effort limits, whether to enforce joint position limits, whether to enforce joint velocity limits, specifying which joints should not be gravity compensated and controlling how much log information is generated during runtime.

15. The computer program product as recited in claim 9, wherein said compound task comprises a plurality of tasks, wherein said plurality of tasks comprises one or more of the following: joint position, two-dimensional orientation, three-dimensional orientation, center of mass, Cartesian position and center of pressure.

16. The computer program product as recited in claim 9, wherein said constraint set comprises a plurality of constraints, wherein said plurality of constraints comprises one or more of the following: contact constraints and transmission constraints.

17. A system, comprising:

a memory unit for storing a computer program for controlling a humanoid robot; and
a processor coupled to the memory unit, wherein the processor is configured to execute the program instructions of the computer program comprising:

defining a compound task that defines operational space objectives and lower priority goal postures that a whole body controller should achieve;

defining a constraint set that specifies natural physical constraints of a robot;

defining a robot model that computes kinematic and dynamic properties of said robot;

defining controller parameters to be utilized by said whole body controller;

binding said controller parameters to a set of transport layers which enables said controller parameters to be accessed by external processes; and

instantiating said whole body controller utilizing said compound task, said constraint set, said robot model and said parameter bindings.

18. The system as recited in claim 17, wherein the program instructions of the computer program further comprise:

defining a hardware abstraction layer comprising a robot interface and a servo clock, wherein said robot interface is responsible for obtaining a joint state of said robot and sending a command from said whole body controller to said robot, wherein said servo clock is responsible for initializing said whole body controller.

19. The system as recited in claim 17, wherein the program instructions of the computer program further comprise:

storing definition of events which contains a logical expression over said controller parameters thereby enabling parameter changes to trigger execution of external processes without the use of polling.

20. The system as recited in claim 17, wherein the program instructions of the computer program further comprise:

utilizing a set of service-based introspection capabilities to enable external processes to query controller properties as it is running.

21. The system as recited in claim 17, wherein said whole body controller is instantiated utilizing said compound task, said constraint set, said robot model and said parameter bindings in a multi-threaded system.

22. The system as recited in claim 17, wherein said controller parameters specify one or more of the following: which groups of joints should be coupled, whether to enforce joint effort limits, whether to enforce joint position limits, whether to enforce joint velocity limits, specifying which joints should not be gravity compensated and controlling how much log information is generated during runtime.

23. The system as recited in claim 17, wherein said compound task comprises a plurality of tasks, wherein said plurality of tasks comprises one or more of the following: joint position, two-dimensional orientation, three-dimensional orientation, center of mass, Cartesian position and center of pressure.

24. The system as recited in claim 17, wherein said constraint set comprises a plurality of constraints, wherein said plurality of constraints comprises one or more of the following: contact constraints and transmission constraints.

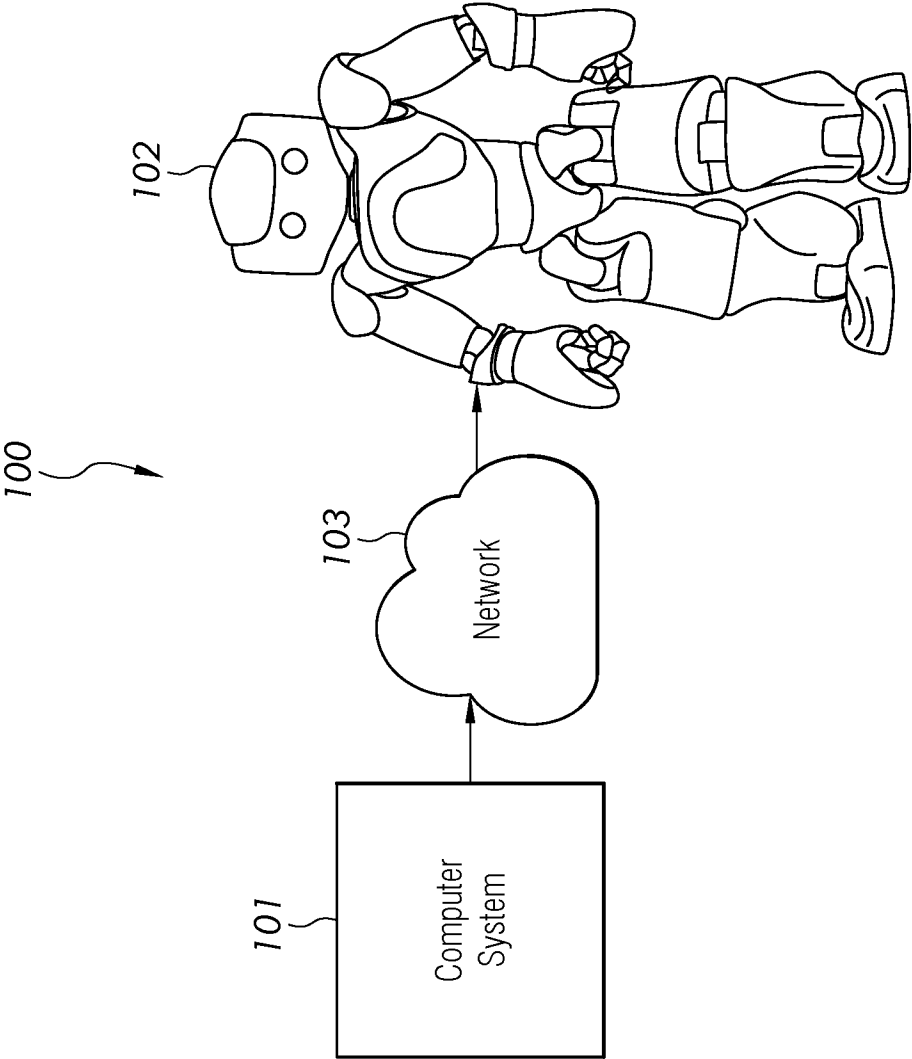


FIG. 1

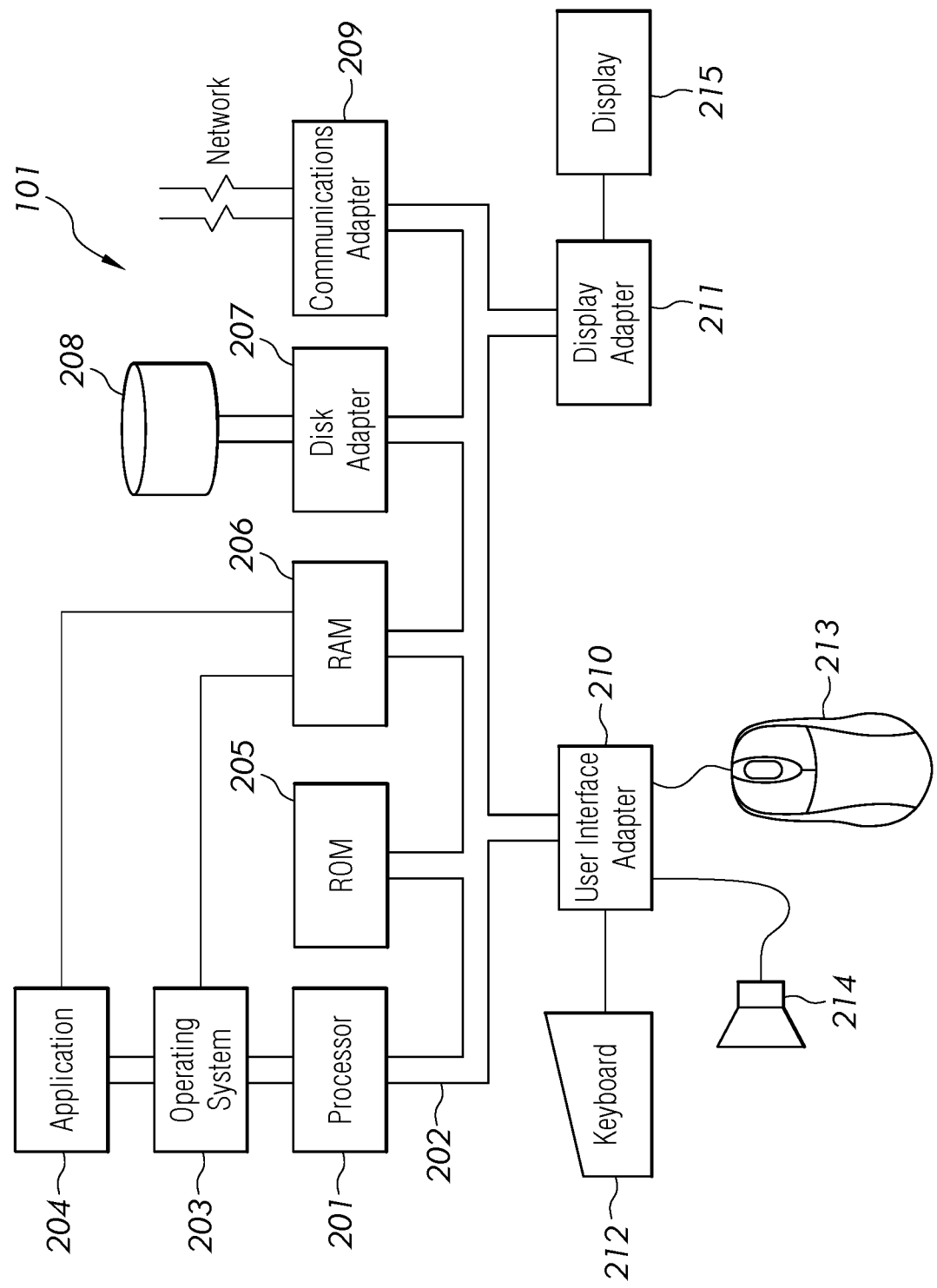


FIG. 2

3 / 15

300

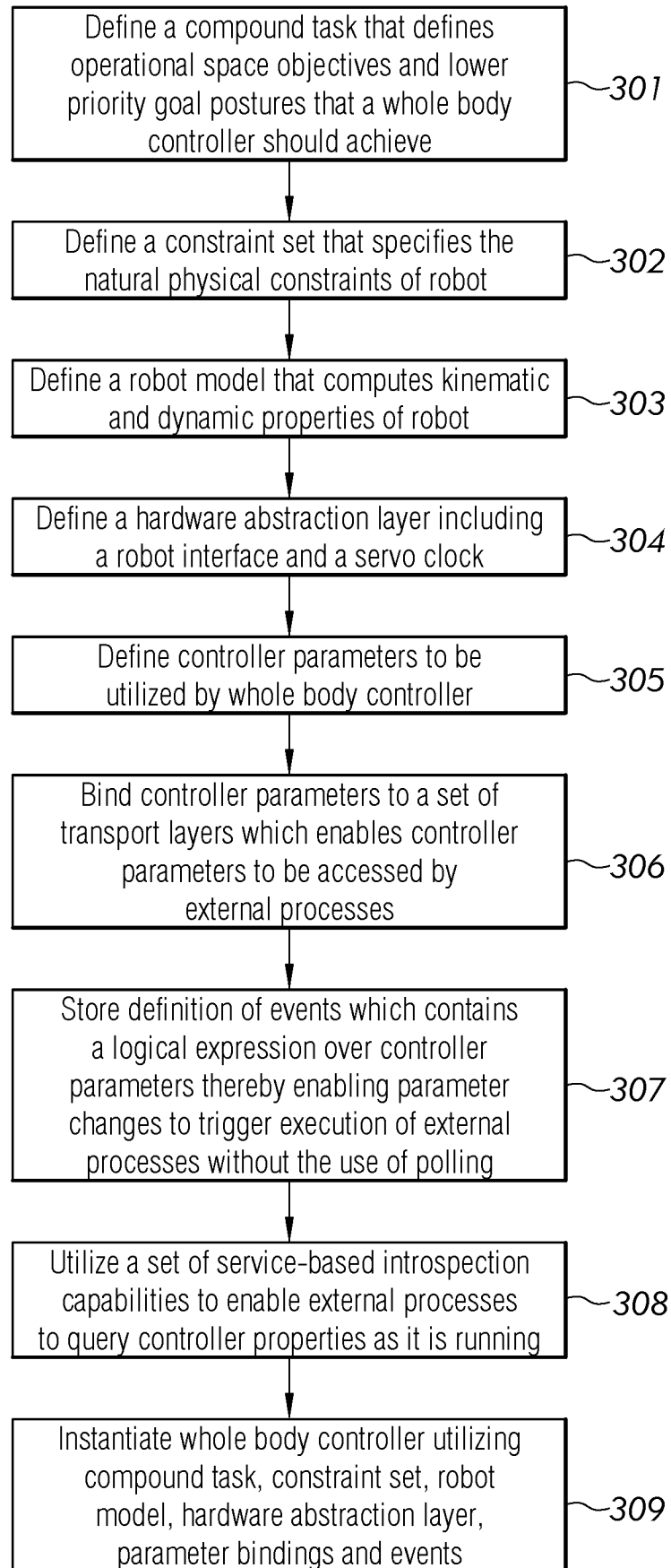


FIG. 3

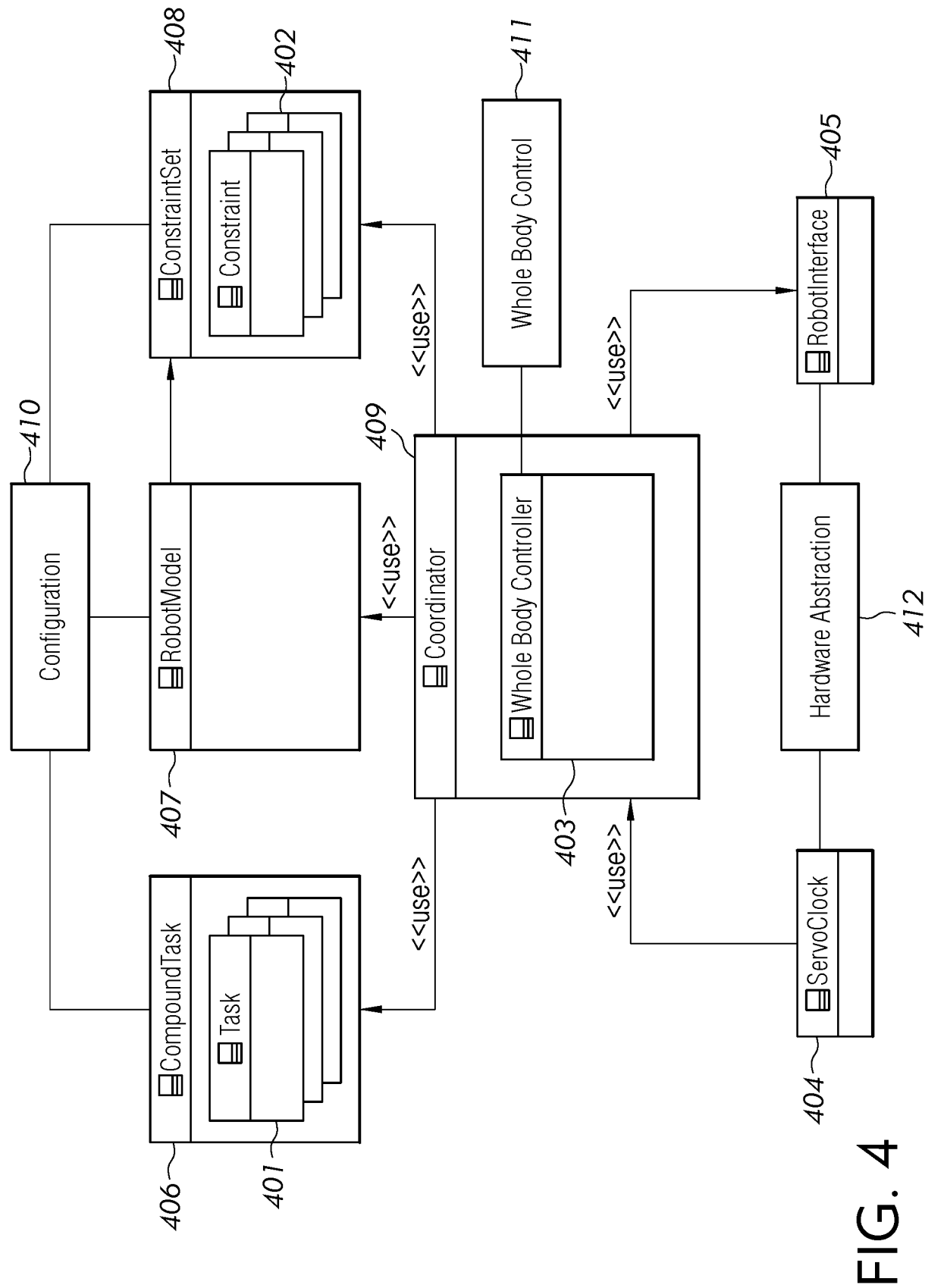
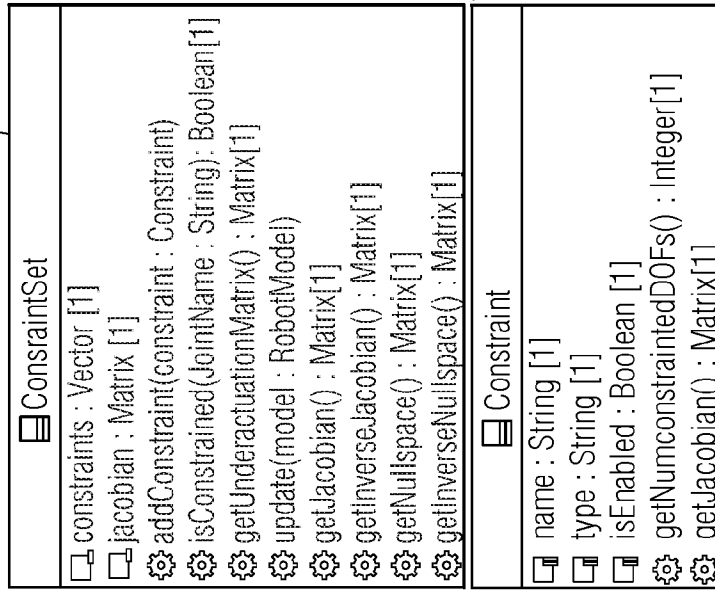


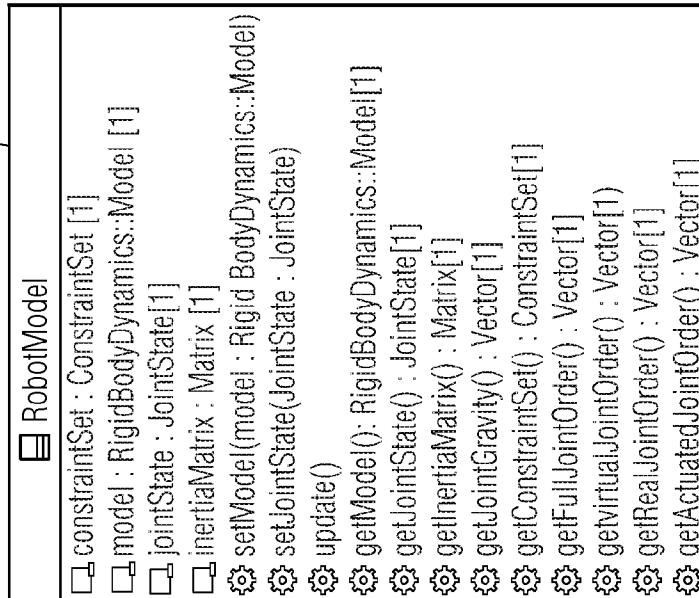
FIG. 4

5 / 15

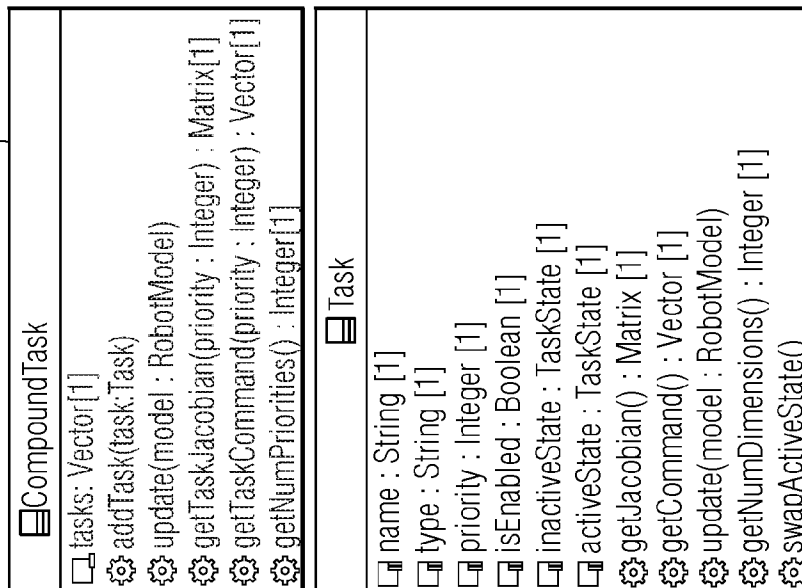
408



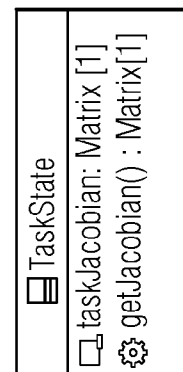
407



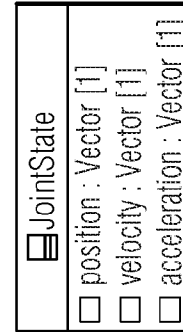
406



401

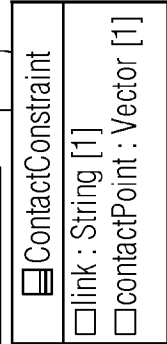


502



501

701



402

705

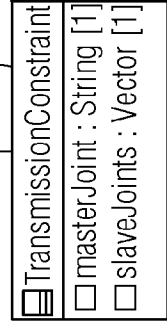


FIG. 5

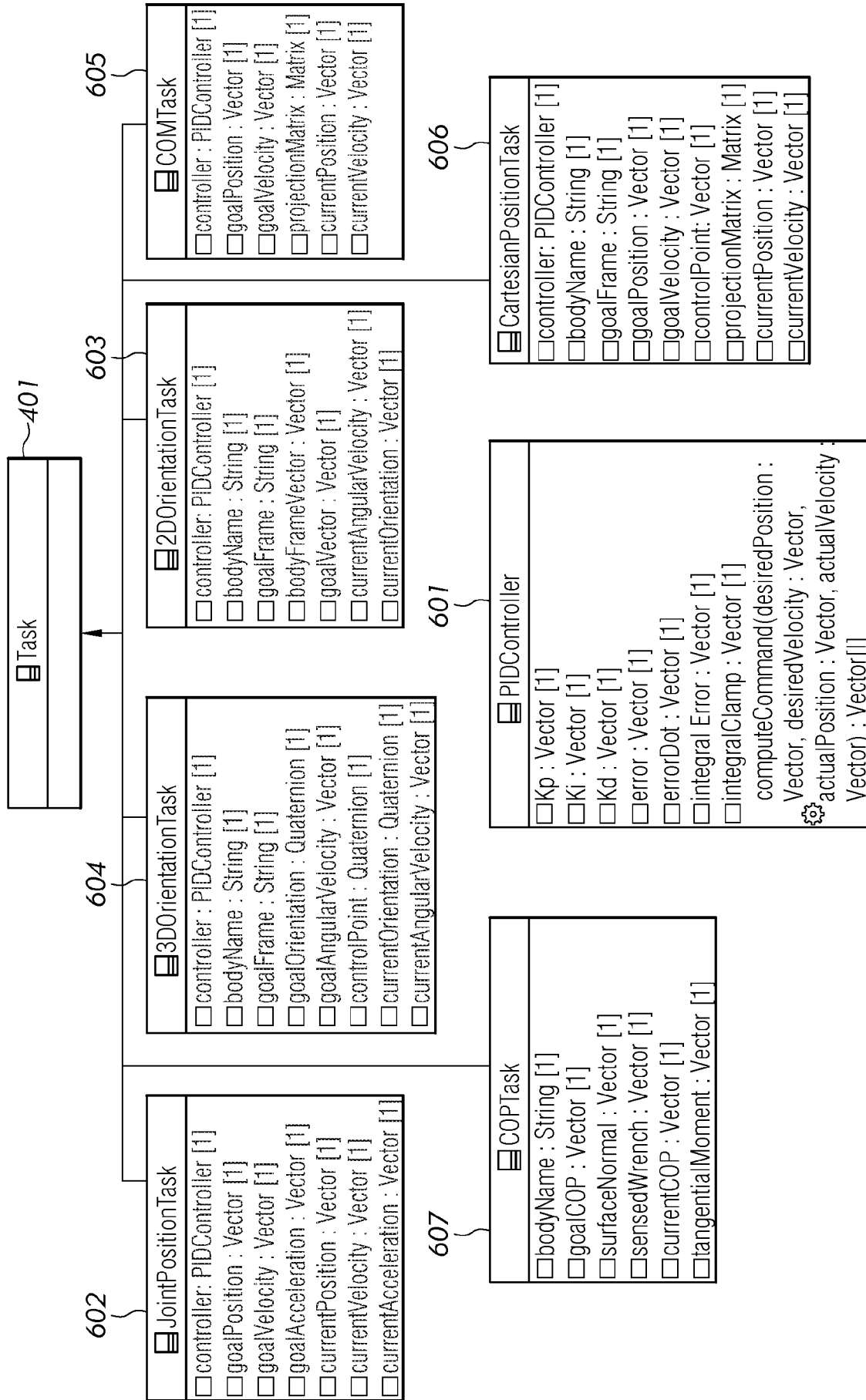


FIG. 6

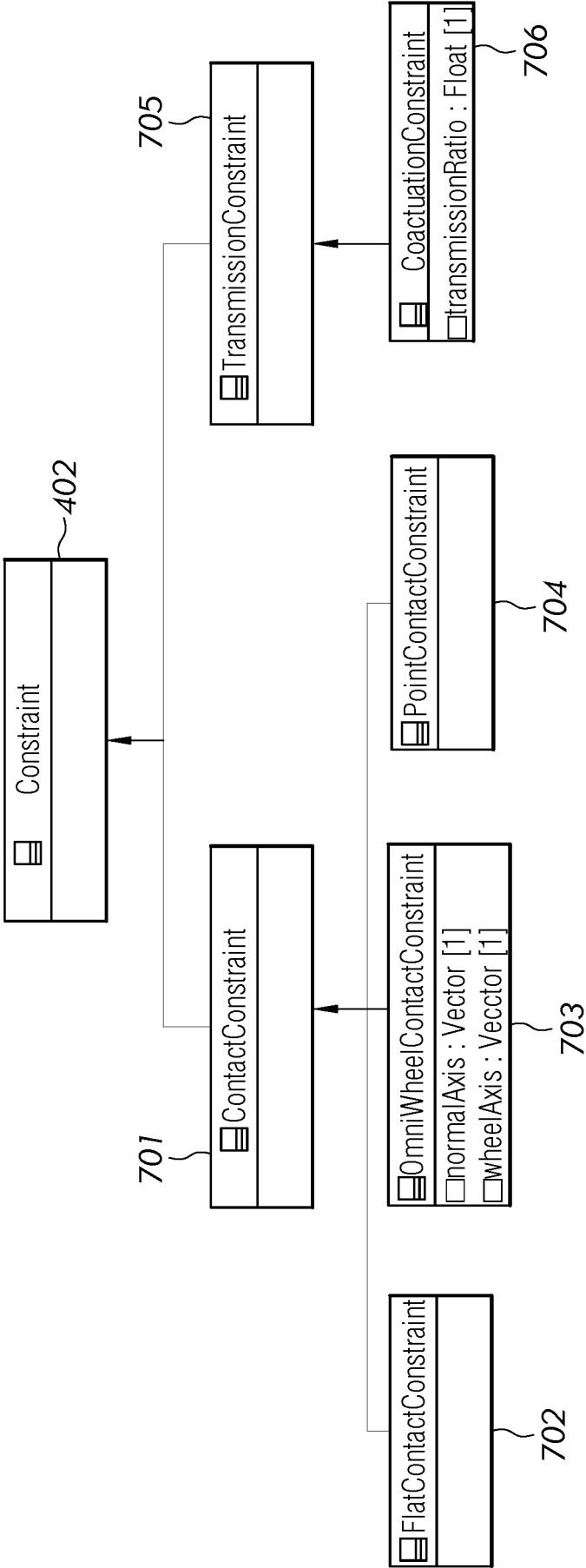


FIG. 7

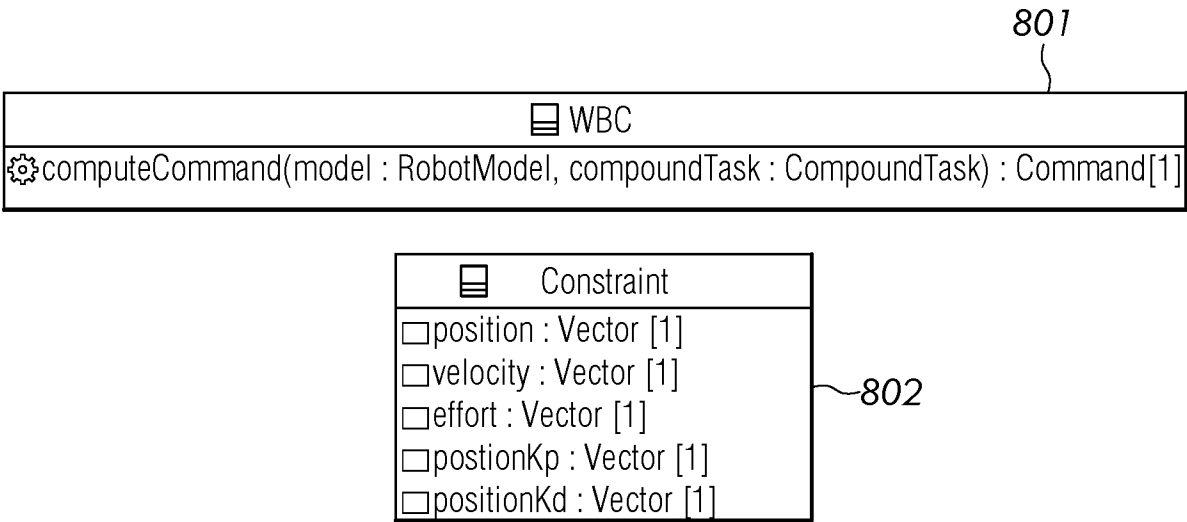


FIG. 8

9 / 15

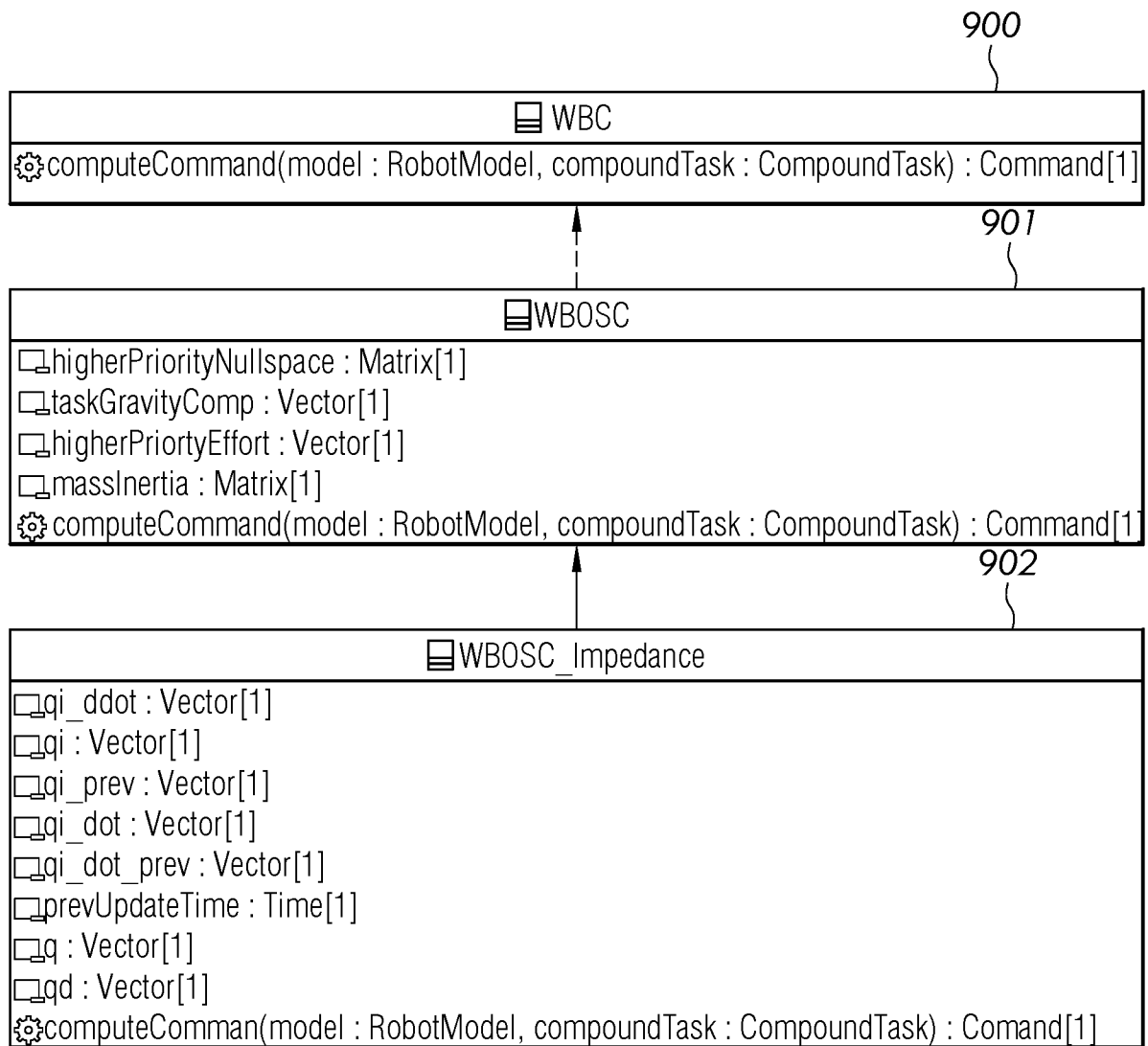


FIG. 9

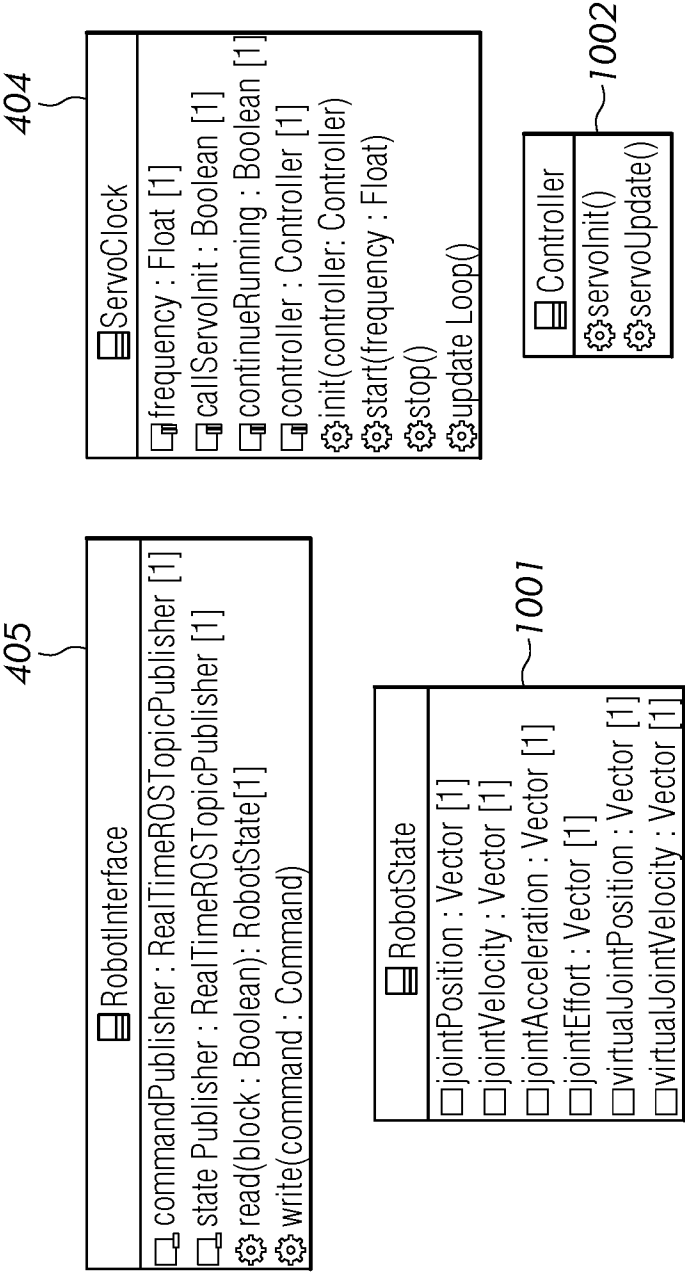


FIG. 10

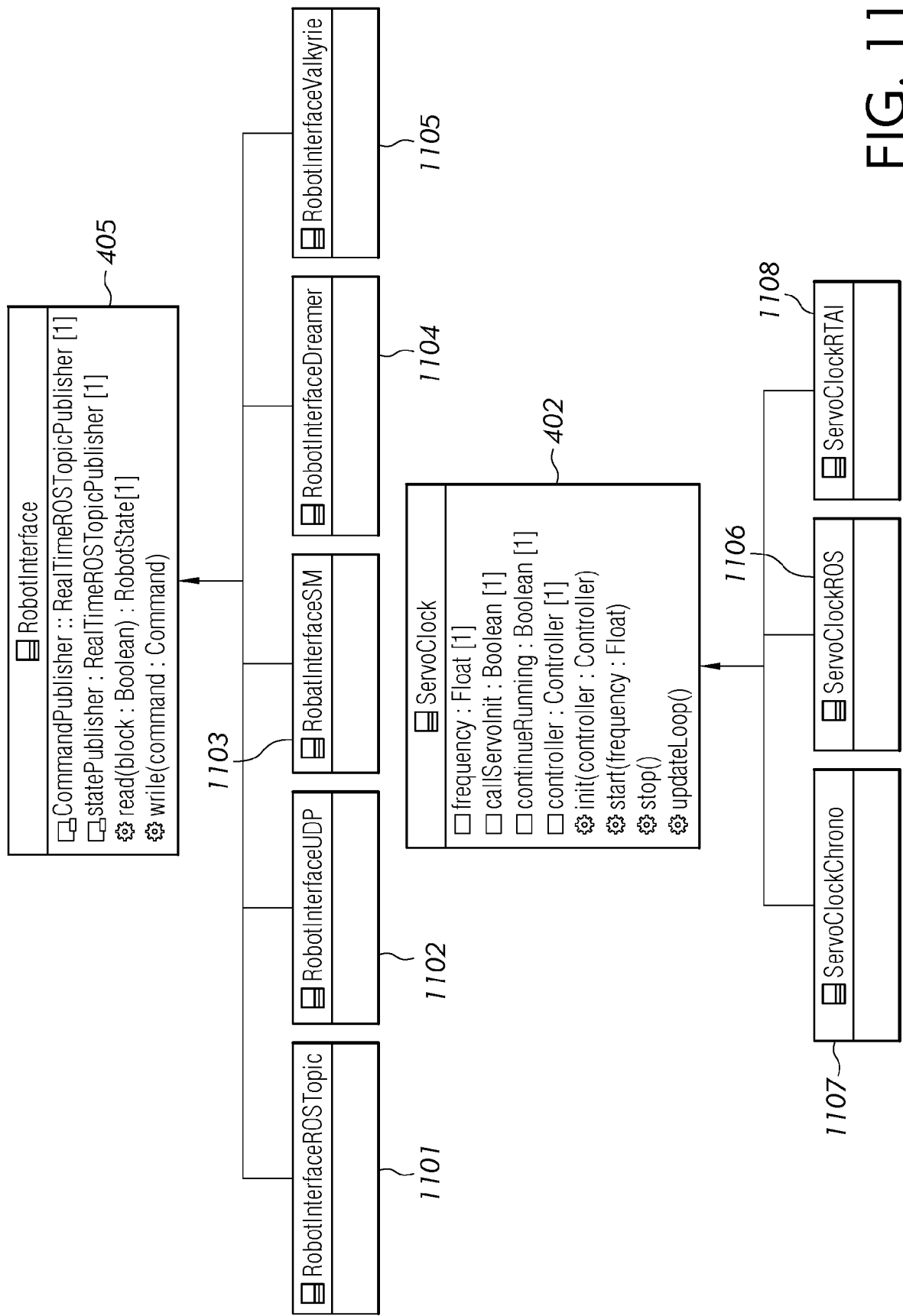


FIG. 11

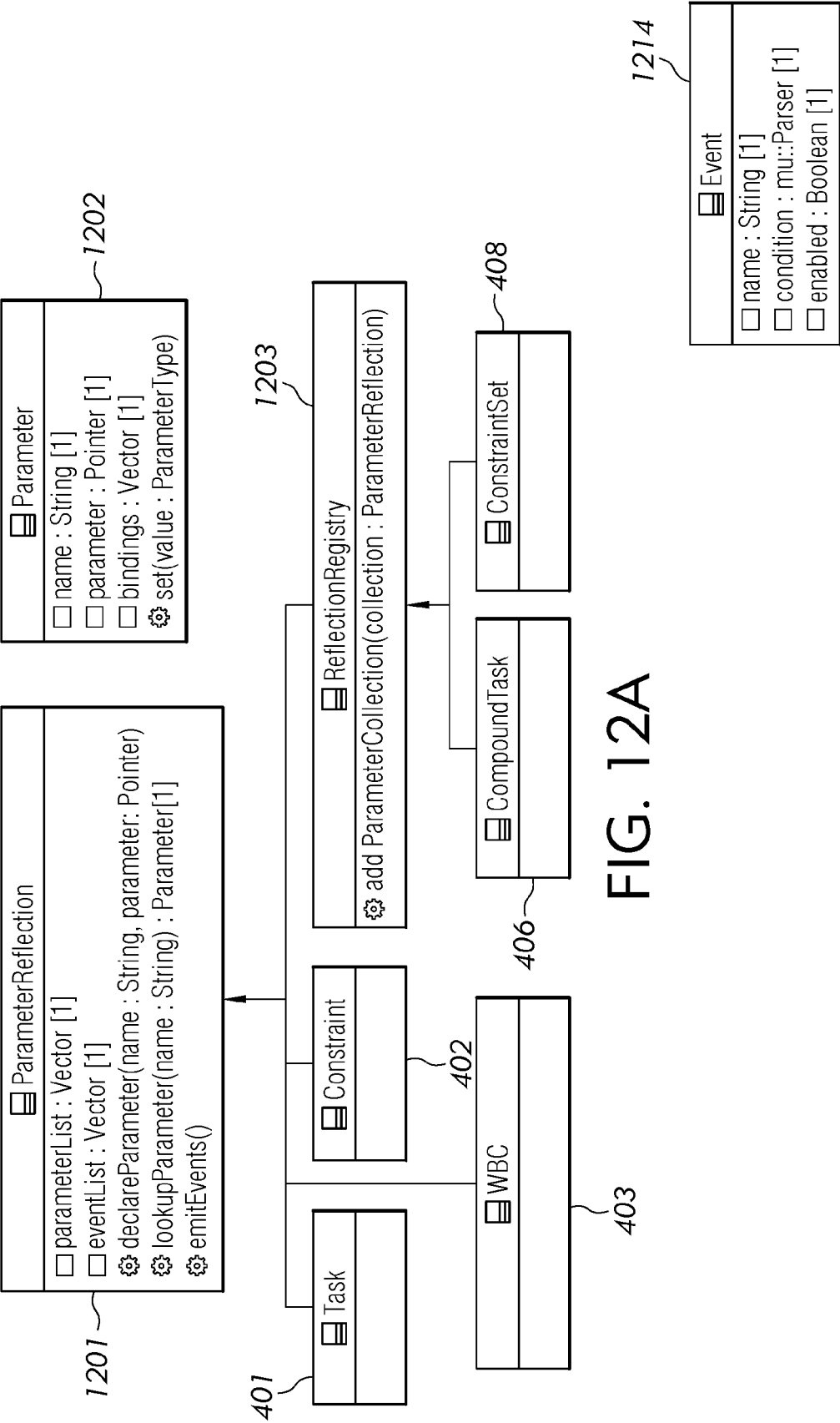


FIG. 12A

FIG. 12C

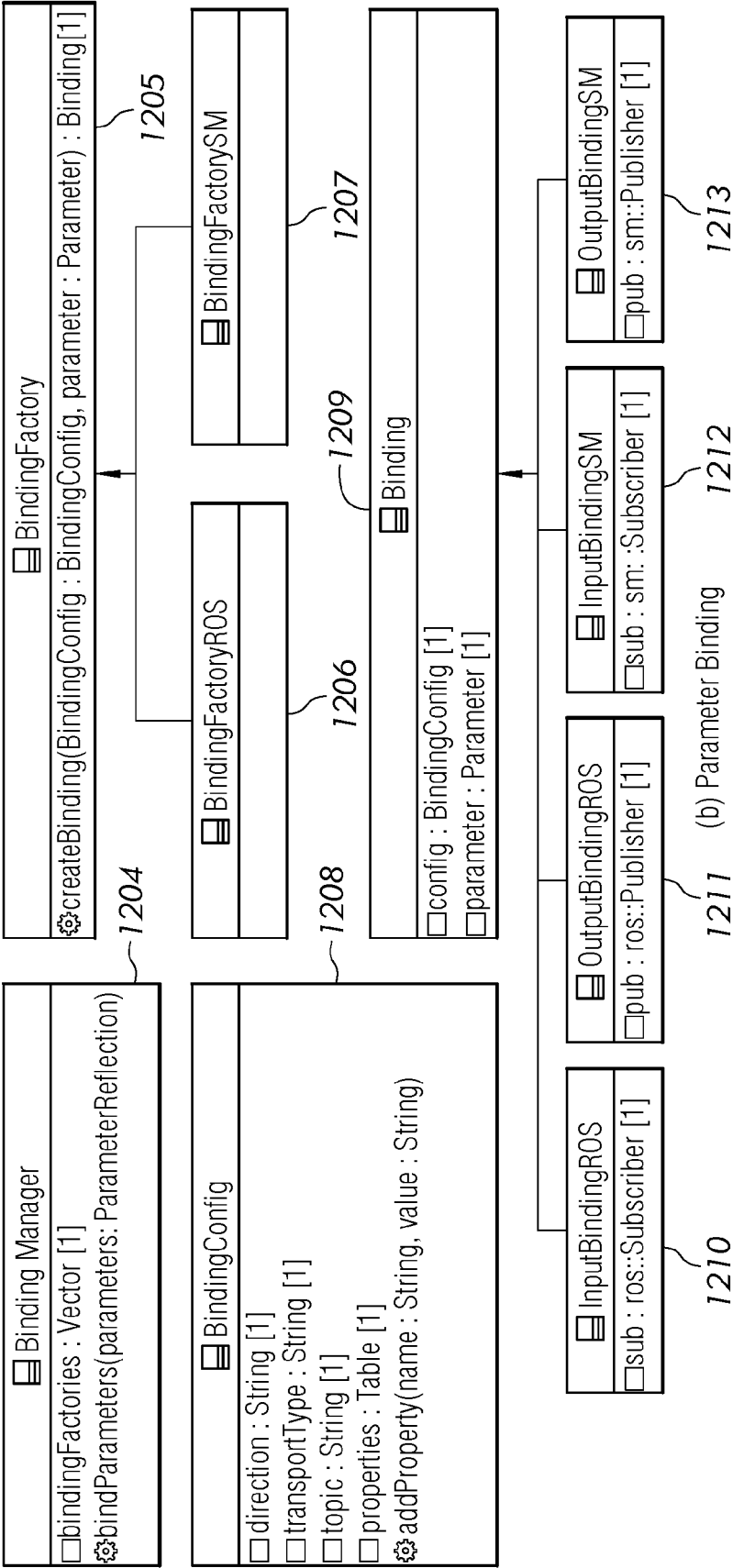


FIG. 12B

14 / 15

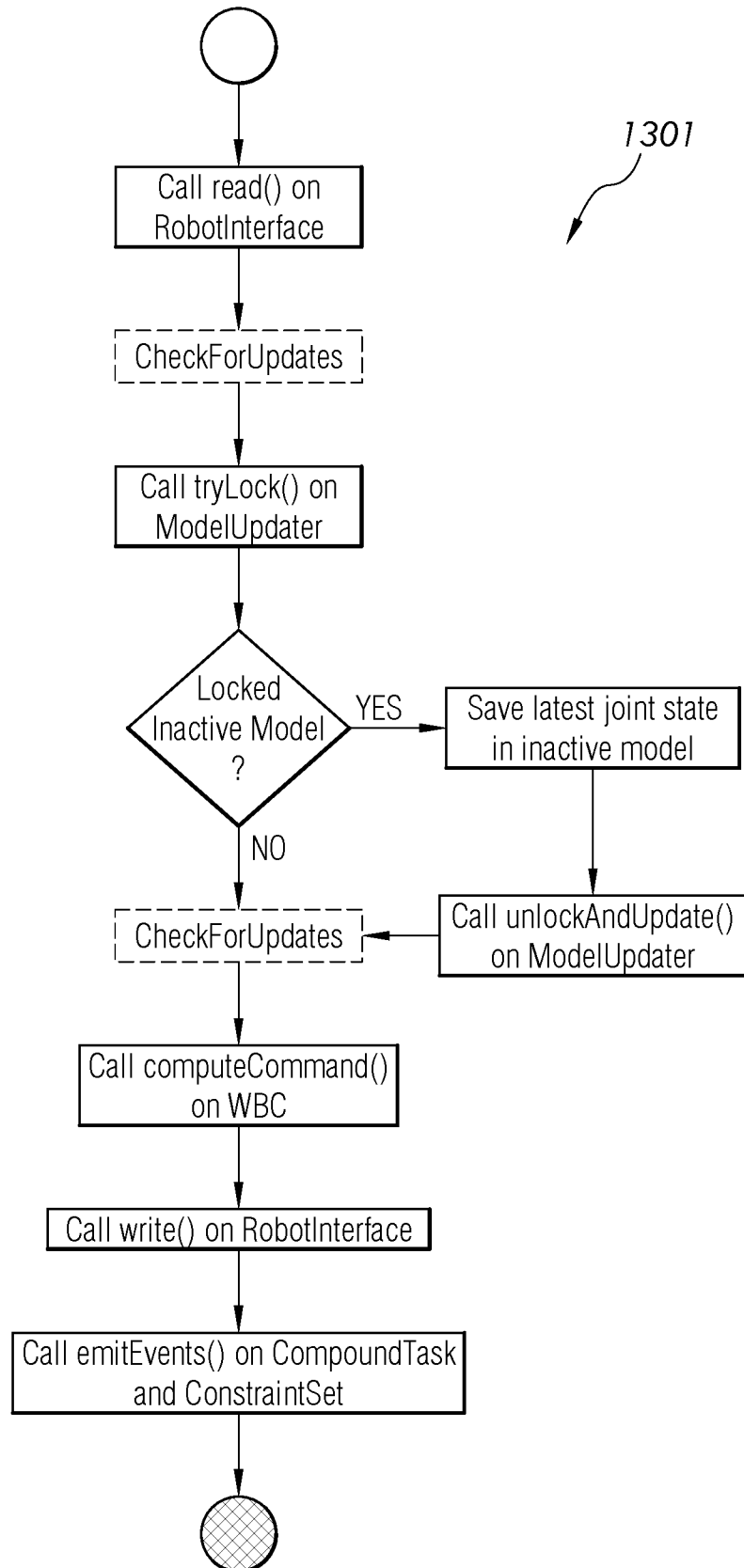


FIG. 13A

15 / 15

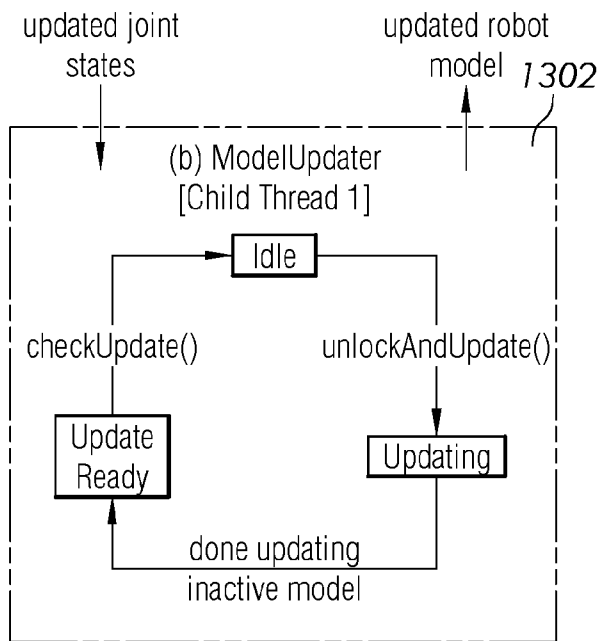


FIG. 13B

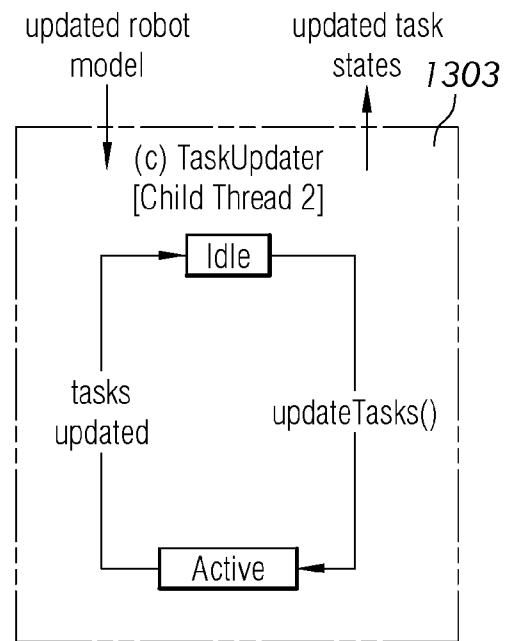


FIG. 13C

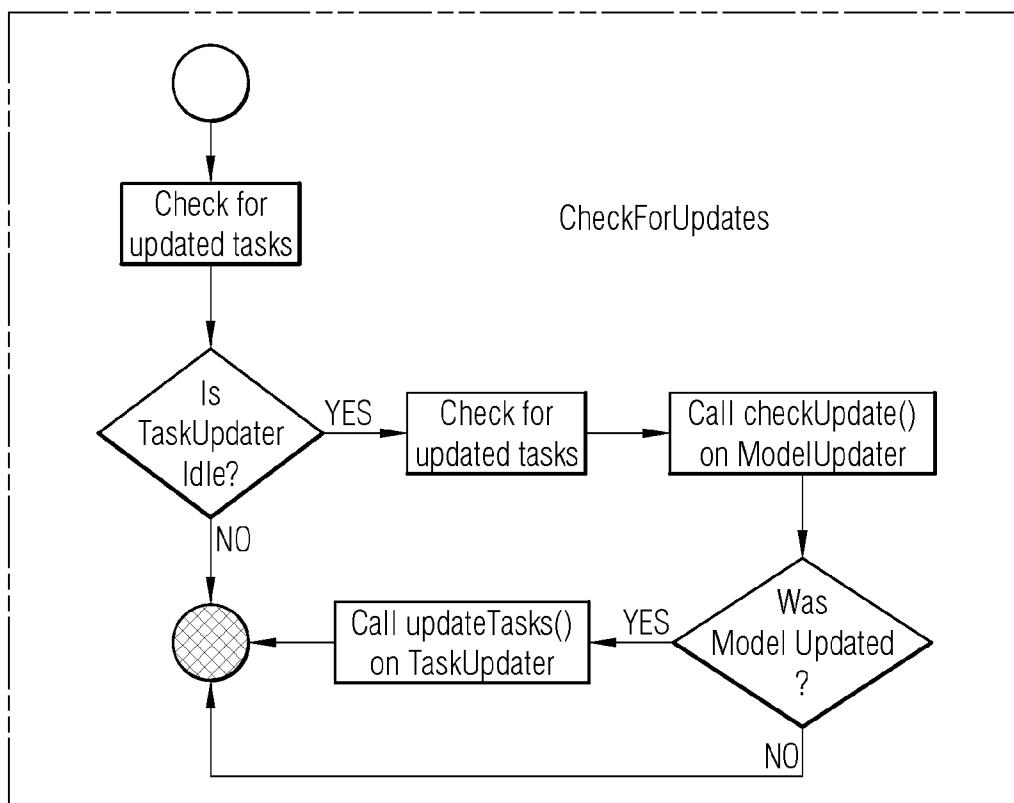


FIG. 13D