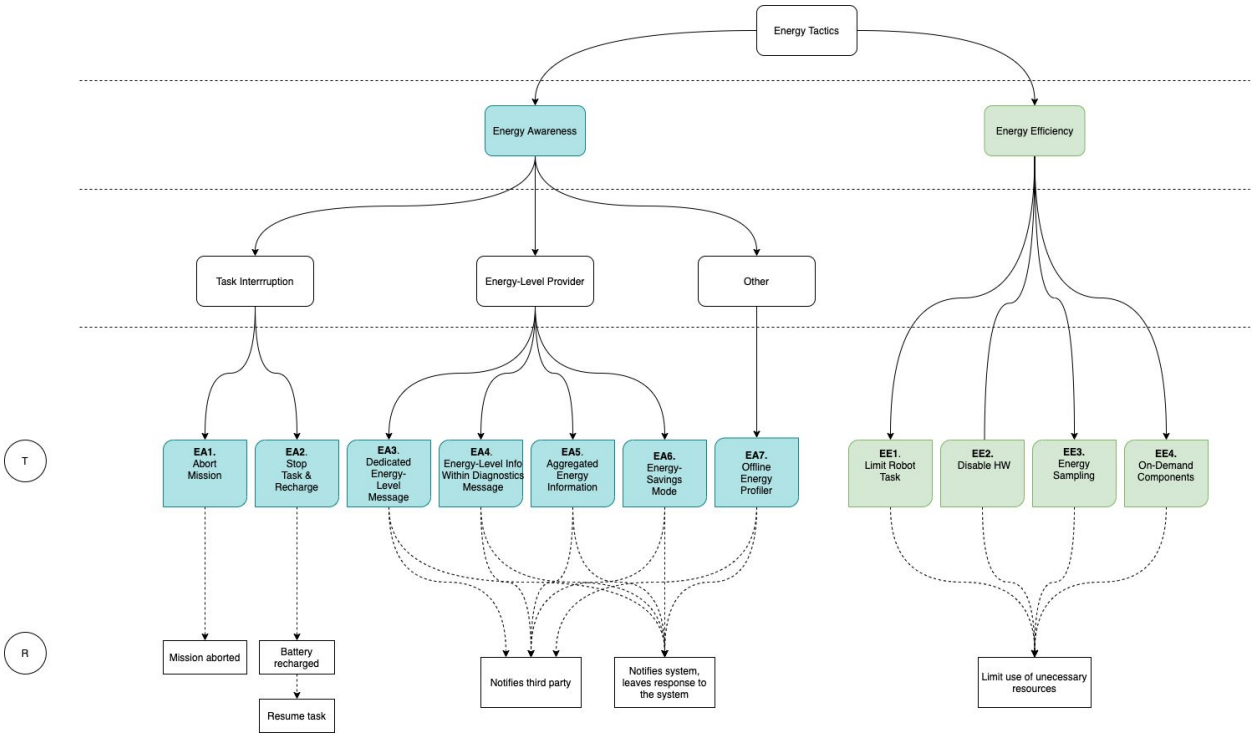
Tactic Tree


Energy Tactics
Energy Awareness
Energy Efficiency
Task Interruption
Energy-Level Provider
Other
T
EA1. Abort Mission
EA2. Stop Task & Recharge
EA3. Dedicated Energy-Level Message
EA4. Energy-Level Info Within Diagnostics Message
EA5. Aggregated Energy Information
EA6. Energy-Savings Mode
EA7. Offline Energy Profiler
EE1. Limit Robot Task
EE2. Disable HW
EE3. Energy Sampling
EE4. On-Demand Components
R
Mission aborted
Battery recharged
Resume task
Notifies third party
Notifies system, leaves response to the system
Limit use of unecessary resources

For all of the following tactics, a component diagram is used to show the different components involved in the system and the interfaces provided and required by the components. A sequence diagram is also included to illustrate the order of interactions between the components and how operations are carried out.

Energy Awareness

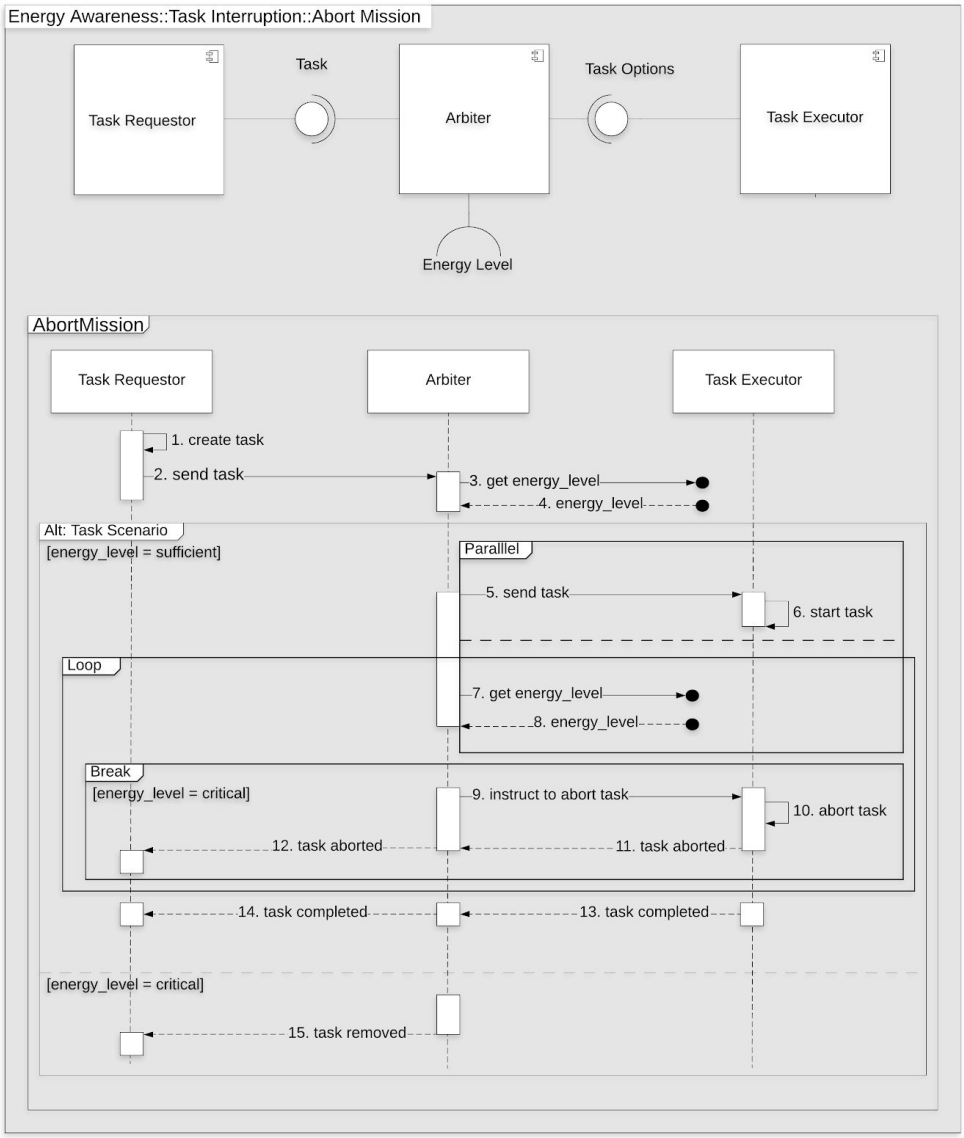
Energy awareness is a crucial tool in robotics software systems as robots are often required to manage their energy wisely. Energy awareness can facilitate robot energy management with the help of different architectural tactics.

EA1: Abort Mission

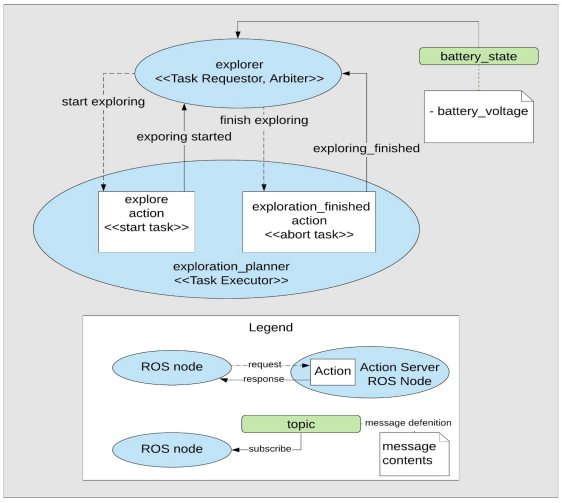
Tactic Name	EA1: Abort Mission
Targeted QA	Energy Awareness
Family	Task Interruption
Motivation	This tactic offers system-wide energy-awareness by monitoring the state of the robot's energy level and ensuring that tasks execute to completion only if the energy levels are enough for the robot to execute them safely.
Description	This tactic aborts a task when the energy-level of the robot reaches a critical point. The *Task Requestor* is responsible for requesting to execute a certain task, the *Arbiter* is responsible for deciding whether or not to abort or execute the task, and the *Task Executo*r is responsible for either aborting or executing the task. First, the *Task Requestor* creates a task and sends it to the *Arbiter* (labels 1,2). After receiving an initial task, the *Arbiter* gets the energy-level of the robot (labels 3, 4); this information is provided to the *Arbiter* by another component in the system. If the energy-level is critical, the *Arbiter i*mmediately removes the task (label 15, the task is not forwarded to the *Task Executor*). In the case when the energy-level is sufficient, in a separate thread, the *Arbiter* sends the task to the *Task Executor* (label 5) and the *Task Executor* starts executing the task (label 6). In another parallel thread, the *Arbiter* starts checking the energy-level within a loop (labels 7,8). If the energy-level remains to be sufficient throughout the entire execution of the task, the *Task Requestor* is notified that the task has been completed (labels 13, 14). If during the execution of the task the energy-level becomes critical, a break statement is issued and the loop is exited (break fragment). The *Arbiter* then instructs the *Task Executor* to gracefully abort the task (labels 9, 10, 11, 12). *The order of the calls in the sequence diagram does not necessarily take place in a chronological order: for example, after label 4, label 15 may take place instead of label 5, depending on the scenario.*
ROS Example	Data point #41: The example below shows how the *Abort Mission tactic* is implemented in an autonomous multi-robot ROS-based system. There are two nodes involved - the *explorer node* and the *exploration_planner node*. The explorer node represents the Task Executor and the Arbiter; it creates a task and is also subscribed to a *battery_state topic* which advertises information regarding the robot's battery such as the battery percentage. The *exploration_planner node* advertises an *explore action* (starts executing a task) and an *exploration_finished action* (aborts a task). The *explorer node* sends a task to the *exploration_planner node*; and instructs the *exploration_planner node* to either start exploring (start task), or finish exploring (abort task), depending on the battery_level. A topic communicates the battery state information to the ROS node as topics are used for one-way continuous data streams. The *explore task* and *exploration_finished task* functionalities are actions, as actions are typically used for any type of behaviour that involves moving the robot, or tasks that run for a longer time. Actions can also be preempted and they satisfy real-time constraints [1].
Other Examples	• A quadrotor (UAV) - dictate to land when the energy levels are critical • Autonomous marine surface vehicle- instruct to power off vehicle when energy levels are critical

[1] http://wiki.ros.org/ROS/Patterns/Communication

Constraints	The tactic as described assumes that the communication between the physical device and the *Arbiter* is already implemented.
Dependencies	This tactic can be combined with tactic EA3 (e.g., provide the energy level in dedicated energy-level component) or EA4 (e.g., include the energy level in some generic component along with other diagnostics information)
Variations	• Event-based logic can be used for checking the energy-level: for example, a global value (e.g., ROS topic) can be accessed at any execution point during runtime. • The *Arbiter* may already check the energy-level in a loop prior to receiving a task.



EA1 Tactic



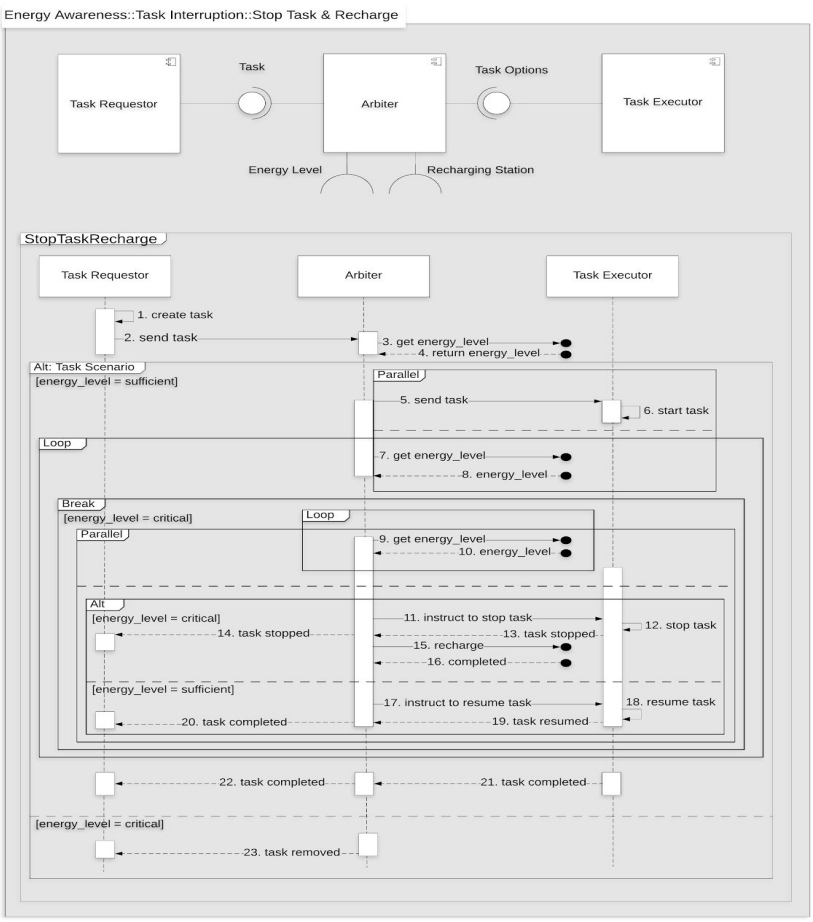
EA1 ROS Example

EA2: Stop Task & Recharge

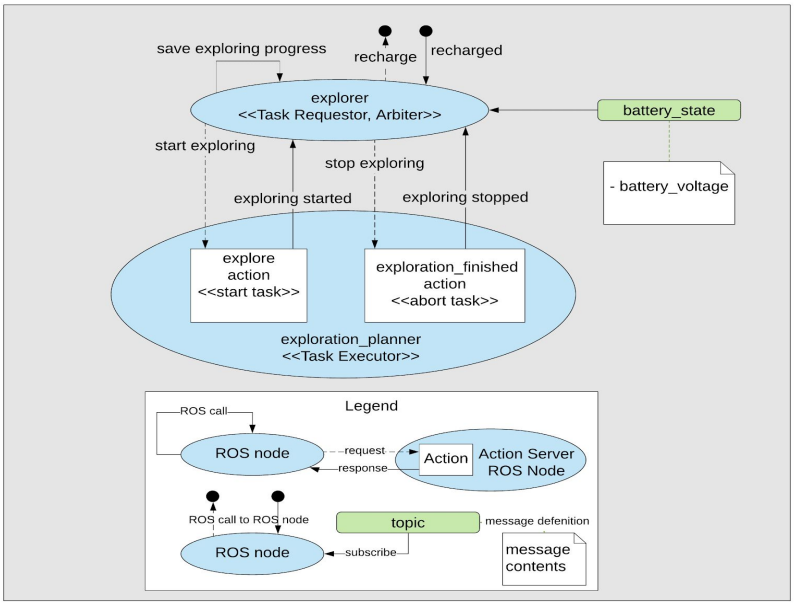
Tactic Name	EA2: Stop Task & Recharge
Targeted QA	Energy Awareness
Family	Task Interruption
Motivation	Ensuring that robots are able to continuously execute a task is an important part of continuous, repetitive, and autonomous robotic systems. Human intervention is not always available when the battery levels are critical, therefore, it is important that the robot is energy-aware, able to replenish its battery power when needed, and is able to safely complete its current mission. This tactic is useful when it is essential that the task is completed.
Description	This tactic gracefully interrupts a task to prevent the robot from fully discharging its battery by instructing it to recharge when the energy level reaches a critical point. The task is resumed when the battery is sufficiently charged. The *Task Requestor* is responsible for requesting to execute a certain task, the *Arbiter* is responsible for deciding whether or not to stop a task and recharge the battery or execute the task, and the *Task Executo*r is responsible for either stopping or executing the task. After creating a task, the *Task Provider* sends the task (label 2) to the *Arbiter* which then checks the energy-level of the robot's battery. If the energy-level is critical, the *Arbiter i*mmediately removes the task (label 23, the task is not forwarded to the *Task Executor*). In the case when the energy-level is sufficient, in a separate thread, the *Arbiter* sends the task to the *Task Executor* (label 5) and the *Task Executor* starts executing the task (label 6). In another parallel thread, the *Arbiter* starts checking the energy-level within a loop (labels 7,8). If throughout the entire execution of the task the energy-level stays sufficient, the *Task Requestor* is notified about the completion of the task (labels 21, 22). If during the execution of the task the energy-level becomes critical, a break statement is issued and the loop is exited (break fragment). A new loop is issued with two parallel threads running inside: one thread where the *Arbiter* checks for the energy-level (labels 9, 10), and another thread where the *Arbiter* decides what to do based on the energy-level. If the energy-level is critical, the *Arbiter* will instruct the *Task Executor* to stop the task (label 11) and request another component in the system to recharge the battery (labels 15, 16). Once the battery is recharged (the energy-level is sufficient), the *Arbiter* instructs the *Task Executor* to resume the task (label 17). If the energy-level drops to a critical point, the previously described steps will take place (labels 11-16).
Constraints	The tactic as described assumes that the communication between the physical device and the *Arbiter* is already implemented.
ROS Example	Data point #24: The example below shows how the Stop Task & Recharge *tactic* is implemented in an

	autonomous multi-robot ROS-based system. There are two nodes involved - the *explorer node* and the *exploration_planner node*. The explorer node represents the *Task Executor* and the *Arbiter*; it creates a task and is also subscribed to a *battery_state topic* which advertises information regarding the robot's battery such as the battery percentage. The *exploration_planner node* advertises an *explore action* (starts executing a task) and an *exploration_finished action* (stops a task). The *explorer node* sends a task to the *exploration_planner node*; and instructs the *exploration_planner node* to either start exploring (start task), or finish exploring (stop task), depending on the battery_level. If the battery level is critical, the *explorer node* sends a request to the *exploration_finished action* to stop the task. It then saves the current progress of the task by calling an internal method. It then instructs another ROS node in the system to recharge the battery. Once the battery is sufficiently charged, the *explorer node* sends a request to the *explore action* to resume the interrupted task by providing the saved task information. A topic communicates the battery state information to the ROS node as topics are used for one-way continuous data streams. The start task and abort task functionalities are actions, as actions are typically used for any type of behaviour that involves moving the robot, or tasks that run for a longer time. Actions can also be preempted and they satisfy real-time constraints[2].
Other Examples	• Autonomous vehicle - instruct vehicle to stop driving and recharge when energy levels are critical. • Robot executing tasks via SMACH (state machine) - store state of containers, and when the battery level is critical, navigate robot to a charging station, and resume current state when the battery is sufficiently charged.
Dependencies	This tactic can be combined with tactic EA3 (e.g., provide the energy level in dedicated energy-level component) or EA4 (e.g., include the energy level in a generic component along with other diagnostics information)
Variations	• Event-based logic can be used for checking the energy-level: for example, a global value (e.g., ROS topic) can be accessed at any execution point during runtime. • The *Arbiter* may already check the energy-level in a loop prior to receiving a task.

[2] http://wiki.ros.org/ROS/Patterns/Communication



EA2 Tactic

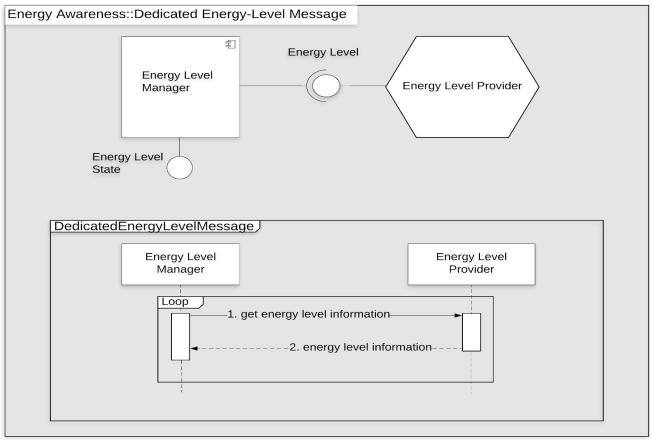


EA2 ROS Example

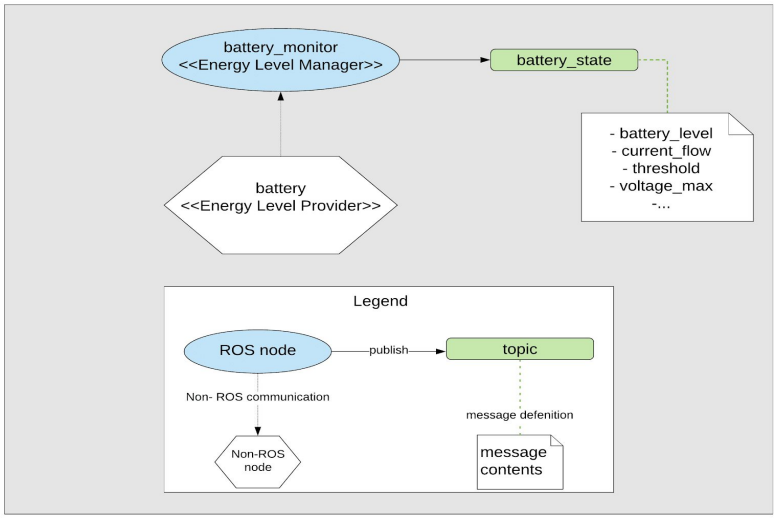
EA3: Dedicated Energy-Level Message

Tactic Name	EA3: Dedicated Energy-Level Message
Targeted QA	Energy Awareness
Family	Energy Awareness
Motivation	The *Dedicated Energy-Level Message* tactic comes in handy when robots require continuous energy-level management as the message communicates information regarding the robot's energy information. This information can be used in several scenarios to promote energy awareness, such as: i) a component that has access to the dedicated energy-level message and uses the information to control the energy-level of a component (e.g., physical battery), ii) without the need for human intervention, a robot can also use the dedicated energy-level message to take certain actions to manage the energy state on its own, and iii) a component which has access to the dedicated energy-level message and uses the energy-level information to notify the user directly. This tactic is ideal when a dedicated single access point for energy-level information is needed.
Description	This tactic monitors the energy level of the robot and provides it in a dedicated message. The *Energy Level Manager* gets the energy level information (label 1) directly from the *Energy Level Provider* (e.g., physical battery) and provides this information in a dedicated *Energy Level State Interface* which can be accessed by other components in the system.
ROS Example	Data Point #22: The figure below illustrates an example of the *Dedicated Energy-Level Message tactic* and how it is employed in a maritime ROS-based system. In this tactic, there is one ROS node and one non-ROS component: the *battery_monitor node* which represents the *Energy Level Manager* (a ROS node which monitors the state of the battery) and the *battery component* - the *Energy Level Provider* (a non-ROS component which represents the physical battery of the robot). The *battery component* is responsible for sending the information regarding the robot's battery via a non-ROS communication method to the *battery_monitor node*. The *battery_monitor node* publishes the received battery information in a *battery_state topic* which can then be subscribed to by any other ROS nodes in the system. A *battery_state topic* is used to communicate battery information to any subscribed ROS node*, as* topics are used for one-way continuous data flows and no feedback from the subscriber node is needed[3]. The battery information is sent to the *energy_level_manager node* via non-ROS communication methods as a non-ROS component represents the physical battery and the communication involves physical signals.
Other Examples	• Energy management in mobile devices - Cinder operating system[4] allows users and applications to control and manage limited device resources such as energy by collecting information about the device's battery. • Energy management for cloud computing[5] - collecting sensor data from multiple locations and managing physical sensor devices.
Constraints	The tactic as described assumes that the communication between the physical device and the *Energy Level Manager* is already implemented.
Dependencies	-
Variations	• The energy level information can be provided by a software component (which is talking directly to the physical battery) • Event-based logic can be used for checking the energy-level: for example, a global value (e.g., ROS topic) can be accessed at any execution point at runtime.

[3] http://wiki.ros.org/ROS/Patterns/Communication
[4] https://sing.stanford.edu/pubs/cstr-10-02.pdf
[5] https://www.researchgate.net/publication/261265938_Total_Energy_Management_System_for_Cloud_Computing/link/54a05bc40cf267bdb9016620/download



EA3 Tactic

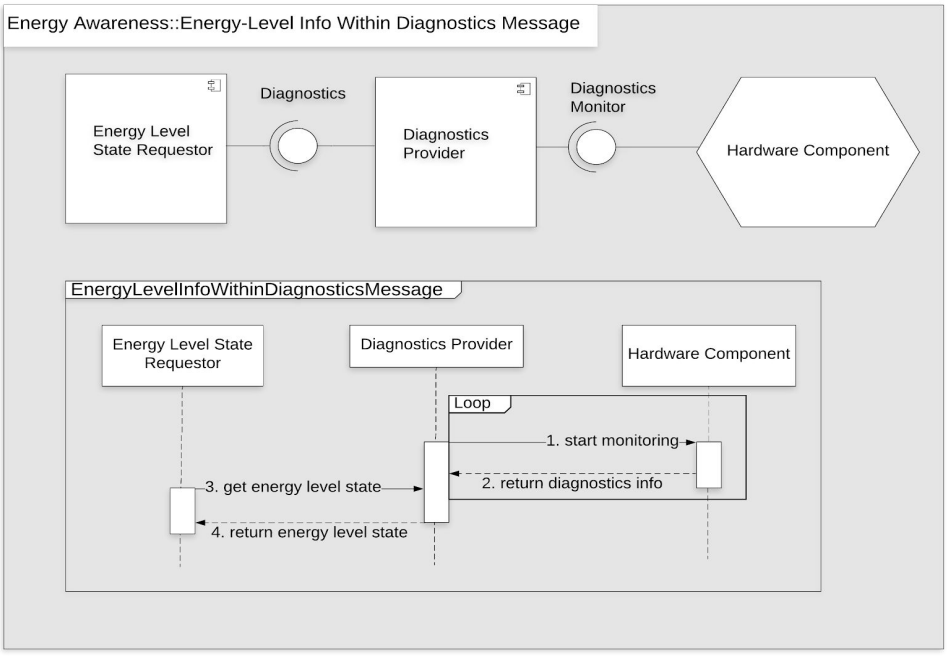


EA3 ROS Example

EA4: Energy-Level Info Within Diagnostics Message

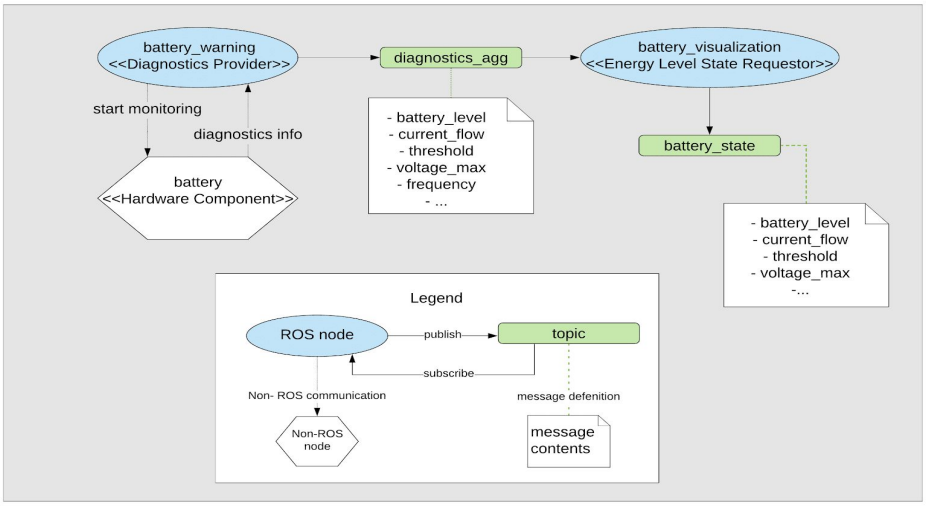
Tactic Name	EA4: Energy-Level Info Within Diagnostics Message
Targeted QA	Energy Awareness
Family	Energy Awareness
Motivation	It is important for the system or user to be aware of the robot's state and ensure that all parts of the robot are running correctly (in addition to the energy level of the robot, information such as frequency, individual hardware component state is important to provide). This tactic is employed when other information (other than the robot's energy-level) regarding the robot's state is important to provide.
Description	This tactic includes the energy level state of a hardware component within a generic diagnostics message. The *Diagnostics Provider* is a generic component which monitors the state of a *Hardware Component* via some monitoring mechanism (labels 1,2) and provides the *Hardware Component* diagnostics information (e.g., connection, power, temperature, warnings) in the *Diagnostics Interface*. The status of the *Hardware Components* energy level is also included in the *Diagnostics Interface* which is requested by the *Energy Level State Requestor* for further analysis (labels 3,4).

ROS Example	Data Point #16: Figure 1(b) is an example of the *Energy-Level Info Within Diagnostics Message tactic* and how it is employed in a humanoid robot via a ROS-based system. In this tactic there are two ROS nodes - the *battery_warning node* (maps to the *Diagnostics Provider*)and the *battery_visualization node* (maps to the *Energy Level State Requestor*). There is also a *battery component* which represents a physical battery. The *battery_warning node* monitors the *battery component*, which provides its diagnostics information. The *battery_visualization requestor node* is subscribed to the *diagnostics_agg topic* which is published by the *battery_warning node* for the battery_level state information, and publishes the received message in a *battery_state topic* which can then be subscribed to by other ROS nodes in the system for visualizing the information. A *battery_state topic* and a *diagnostics topic* are used to communicate battery information to any subscribed ROS node*, as* topics are used for one-way continuous data flows and no feedback from the subscriber node is needed[6]. The battery information is sent to the *diagnostics_provider node* via non-ROS communication methods as a non-ROS component represents the physical battery and the communication involves physical signals.
Other Examples	• Diagnostics in mobile phones - built in diagnostics tools are used for running tests to check things like the device's touch screen, audio, video, camera, microphone, and other components of the phone - the diagnostics information of the mobile device (e.g, frequency, battery state, sensor state) is used to promote energy-awareness • On-board diagnostics (OBD) - self-diagnostic and reporting capability of a vehicle. Provides access to the diagnostics information of different vehicle subsystems (e.g., airbag system, radio system, sensors) to the user.
Constraints	The tactic as described assumes that the communication between the *HW Component* and the *Diagnostics Provider* is already implemented.
Dependencies	-
Variations	• Event-based logic can be used for monitoring the *Hardware Component*: for example, a global value (e.g., ROS topic) can be accessed at any execution point during runtime.



EA4 Tactic

[6] http://wiki.ros.org/ROS/Patterns/Communication



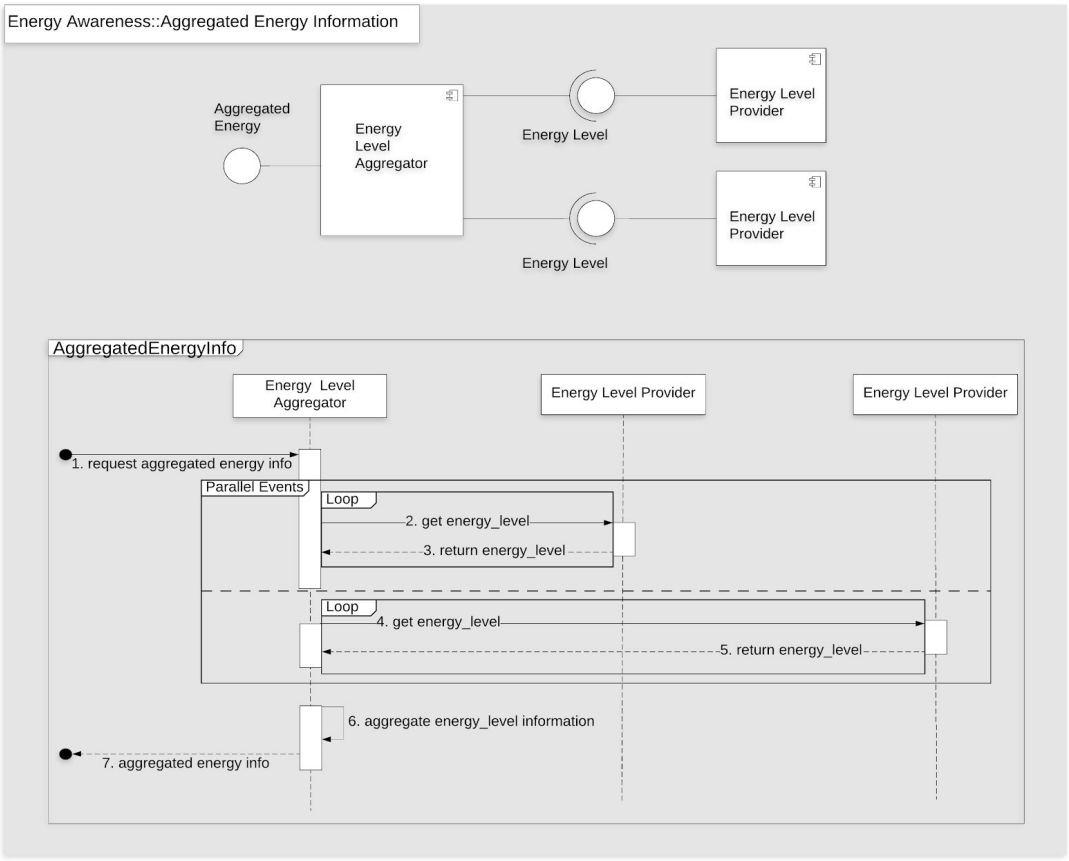
EA4 ROS Example

EA5: Aggregated Energy Information

Tactic Name	EA5: Aggregated Energy Information
Targeted QA	Energy Awareness
Family	Energy Awareness
Motivation	It is important for the robot to be aware of the energy state of its different components. Gathering the energy state of different components and aggregating it makes it easier for the robot or user to be aware of the energy state of the different components, identify patterns and trends that were not visible before for data analysis, and provide a single access point for other software components in the system to access the components' energy state data.
Description	This tactic aggregates energy-level information (e.g., average charge, battery capacity, voltage) of different components (e.g., batteries) into a single interface. The *Energy Level Providers* represent different components of a robot which provide their energy state. It is not limited to only two *Energy Level Providers*; there can be more *Energy Level Providers* in the system. First, an initial request to the *Energy Level Aggregator* is made by another component in the system to get the aggregated energy information (label 1). Then, the *Energy Level Aggregator* gets the energy state of the different *Energy Level Providers* in two parallel separate threads (labels 2-5) and aggregates the data (label 6). The aggregated data is then returned to the requestor (label 7).
ROS Example	Data Point #40: The figure below illustrates an example of the *Aggregated Energy Information tactic* and how it is used in ROS drivers for controlling Segway-based robots in a ROS-based system. In this tactic there are three ROS nodes - the *arduino_driver node* (maps to the *Energy Level Aggregator*) and the *battery_diagnostics nodes* (maps to the *Energy Level Providers,* there can be more than two *battery_diagnostics nodes* in the system). The *battery_diagnostics nodes* (e.g., batteries) publish energy-related information in the separate *battery_state topics*. The *arduino_driver node* is subscribed to these topics and aggregates the energy-related messages into a single message - it then publishes the message to the *diagnostics topic*. This topic can be later subscribed to by other nodes in the system. An *energy_level topic* and a *aggregated_energy topic* are used to communicate battery information to any subscribed ROS node, *as* topics are used for one-way continuous data flows and no feedback from the subscriber node is needed[7].

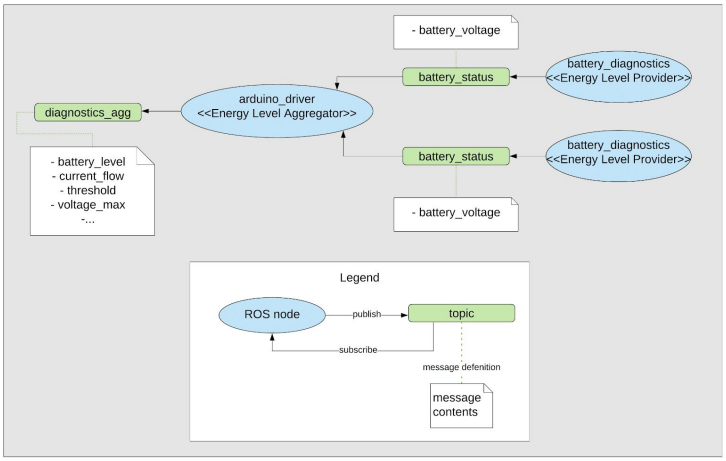
[7] http://wiki.ros.org/ROS/Patterns/Communication

Other Examples	• Microgrid community - peer-to-peer energy-sharing method[8]; aggregated control of many small-scale batteries to manage energy requirements of the entire community.
Constraints	It is assumed that the *Energy State Providers* have an established connection with the physical components in the system.
Dependencies	-
Variations	• Event-based logic can be used for getting the energy-level information: for example, a global value (e.g., ROS topic) can be accessed at any execution point at runtime.



EA5 Tactic

[8] https://www.sciencedirect.com/science/article/pii/S1876610218300845

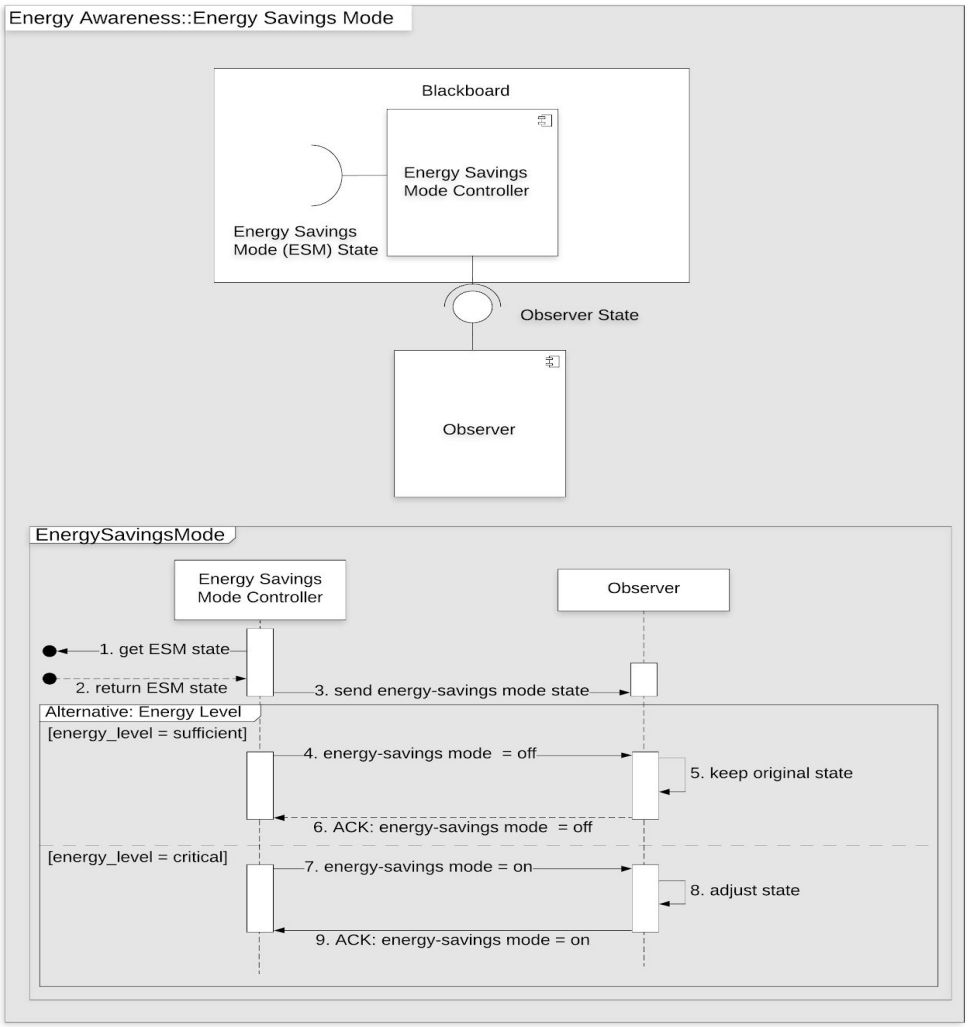


EA5 ROS Example

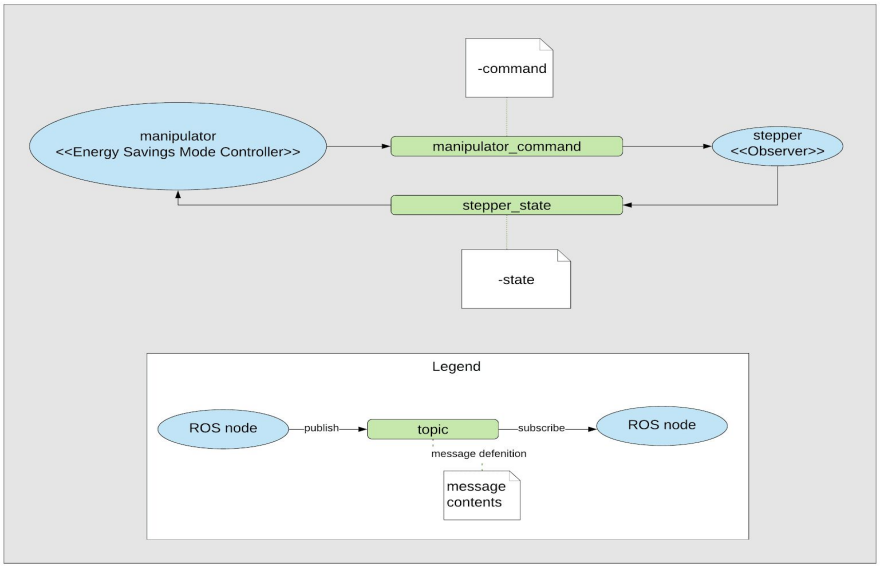
EA6: Energy-Savings Mode

Tactic Name	EA6: Energy Savings Mode
Targeted QA	Energy Awareness
Family	Energy Awareness
Motivation	To ensure that all components are energy-aware, know when they need to start saving energy, and adjust their behaviour accordingly, a shared space for storing the information about the robot's state (e.g., blackboard) is needed.
Description	This tactic dictates to the components in a system whether or not to enter into a state in which energy must be saved (enable or disable the energy-savings mode). The *Energy Savings Mode Controller* acts as a *blackboard* (decentralized) that shares the current energy-savings mode state (on/off) with the rest of the components in the system. The *blackboard* requests the current energy-savings mode state from another component in the system, and based on the response, it dictates to the rest of the components to either disable or enable the energy-savings mode and change their state accordingly. Every component which is represented as the *Observer* component receives the current energy-savings mode state, switches to it, and updates the *blackboard* by sending an acknowledgement message. With this approach, all of the *Observers* are aware of the current energy-savings mode state of the system, and the *blackboard* is aware of whether or not each component switched to the instructed energy-savings mode state.
ROS Example	Data Point #15: Figure 1(b)illustrates an example of the *Energy-Savings Mode tactic* and how it is employed in an ROV via a ROS-based system. There are two nodes involved in this tactic - the *stepper node* (maps to the *Observer*) and the *manipulator node* (maps to the *Energy Savings Mode Controller*). The *stepper node* represents nodes in the system which must adhere to a certain behaviour depending on the current manipulator command (e.g.c current energy-savings mode). The *manipulator node* publishes the current command to a *manipulator_command topic* which is subscribed to by the *stepper node*. After receiving the current manipulate command and adhering to it, the *stepper node* publishes its new state to the *stepper_state topic,* which is in turn subscribed to by the *manipulator node*. With this kind of structure, the *manipulator node* can keep track of whether or not all nodes in the system behave according to the current issued command. An *energy_savings_mode_state topic* and an *observer_state topic* are used to communicate messages between the *energy_savings_mode_controller node* and the *observer node, as* topics are used for one-way continuous data flows.

Other Examples	• Mobile computing - e.g., power-saving mode (PSM) for mobile computing in WiFi-hotspots • IoT - e.g., solar energy for IoT devices that are exposed to sunlight • Appliances such as ovens, microwaves - e.g., turn off the display when in energy-savings mode • Hybrid vehicles - increase fuel efficiency by adding a battery-powered electric motor. When the hybrid auto is in "idle"mode (e.g., stops at a stoplight), the engine is turned off, and the battery-powered electric motor still provides accessories such as audio, air conditioning, etc. • IG-L - regulates CO2 emission by adjusting the digital speed signs on the highways to a lower rate. • Industrial emissions within the EU - industrial companies are able to lower their carbon footprint at moments where it will be increased significantly.
Constraints	It is assumed that the criteria for enabling or disabling the energy-savings mode is already included in the *Energy Savings Mode Controller*. It is also assumed that the communication between the *Energy Savings Mode Controller* and the rest of the component in the system already exists and that the components know how to behave when the energy-savings mode is on/off.
Dependencies	-
Variations	• Pull-based approach: based on the energy-level, the Blackboard maintains the state of the energy-savings mode, and the components in the system request the mode.



EA6 Tactic

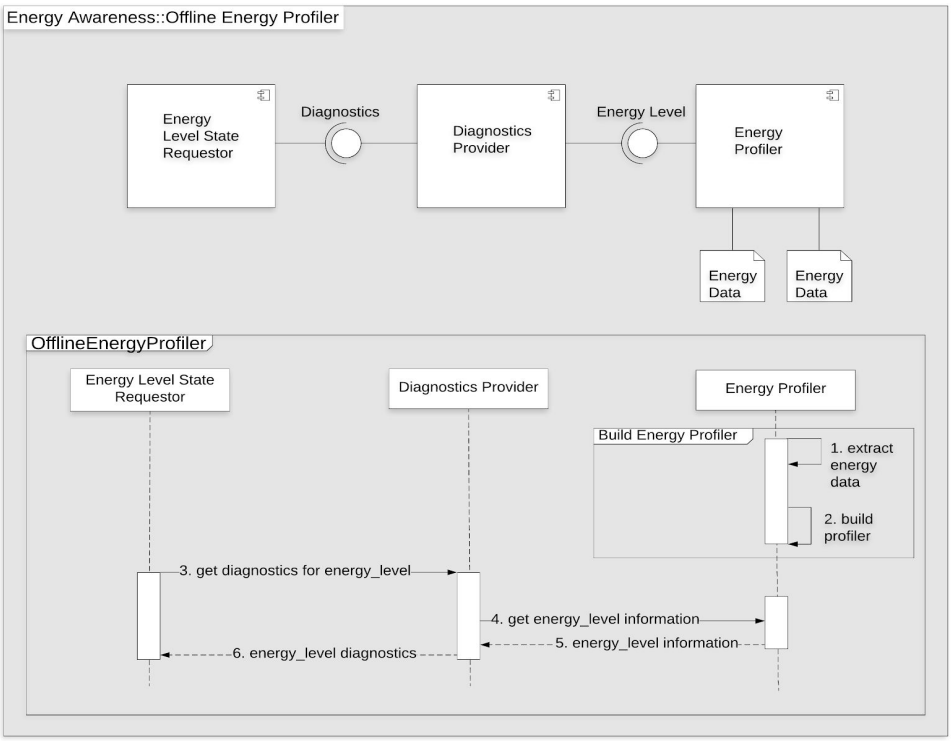


EA6 ROS Example

EA7: Offline Energy Profiler

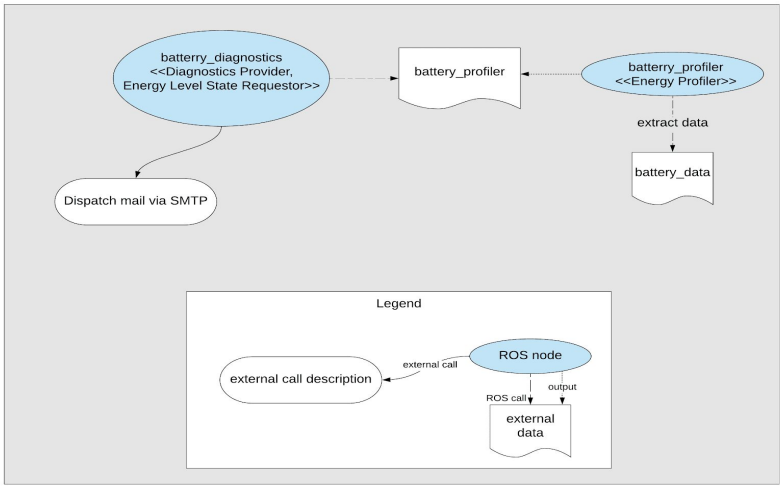
Tactic Name	EA7: Offline Energy Profiler
Targeted QA	Energy Awareness
Family	Energy Awareness
Motivation	Offline profiling provides multiple benefits for robotics developers and researchers such as recording profiled datasets, visualizing and labeling them, and storing them for future use. Besides the previously listed benefits of offline profiling, offline energy profiling also promotes energy-awareness by providing energy diagnostics information from the previous executions to the current execution. For example, this information can be used for estimating the remaining operating time of a battery. For robotics developers and researchers, the recorded energy datasets can be used to analyze trends, patterns, and provide insights on which robotic activities consume the most energy.
Description	This tactic builds an energy profiler from previous executions which is then used in a current execution for providing energy level state diagnostics. The *Energy Profiler* builds an energy profile by extracting logged energy-related data (labels 1,2) (e.g., energy-related messages logged in ROS bag files) and providing its estimations to other components in the system. The *Energy Level State Requestor* requests energy-level diagnostics information from the *Diagnostics Provider* (label 3). After receiving the request, the *Diagnostics Provider* opens an existing energy-profiler and gets the requested energy-level diagnostics information (label 4). This information is then sent back to the *Energy Level State Requestor* (labels 5,6).
ROS Example	Data Point #13: The figure below illustrates an example of the *Offline Energy Profiler tactic* and how it is employed in ROS drivers for controlling Segway-based robots in a ROS-based system. There are two nodes involved in this tactic: a *battery_profiler node* (represents the *Diagnostics Provider* and *Energy Level State Requestor*), and a *battery_diagnostic node* (represents the *Energy Profiler*). The *battery_profiler node* is in charge of building a battery-profiler based on logged battery data from previous executions (e.g., bag files). The *battery_diagnostics node* requests battery diagnostics information (e.g., battery life estimation) by opening an existing battery profiler. It then performs some computation and dispatches the result to a local SMTP client. A *bag* is used to represent the logged energy data as bags are the primary mechanism in ROS for data

	logging[9].
Other Examples	• Silicon Labs - multi-node energy profiler[10]: several nodes monitor different devices, build energy profilers and use them in offline mode for visualizations, computations (e.g., battery estimation), etc. • ALEA: A Fine-Grained Energy Profiling Tool[11]: fine-grained energy profiling tool based on probabilistic analysis for fine-grained energy accounting. An online module transfers the results of profiling to an offline module that derives energy estimates.
Constraints	This tactic as described assumes that the energy profiler exists.
Dependencies	
Variations	• In the case when the energy profiler does not exist, error checking is applied when opening the energy profiler.



EA7 Tactic

[9] http://wiki.ros.org/Bags
[10] https://www.silabs.com/documents/public/user-guides/ug343-multinode-energy-profiler.pdf
[11] https://www.lancaster.ac.uk/staff/wangz3/publications/aleataco.pdf



EA7 ROS Example

Energy Efficiency

As of now, batteries (e.g., Lithium Ion, Lithium Polymer) are heavy to carry and have a limited energy budget; therefore, preserving the robot's battery life is a critical task in robotics software. The following tactics are different ways of how components in the system behave when the energy-savings mode is enabled. The following tactics implement tactic EA6 (the behaviour of the components under the energy-savings mode).

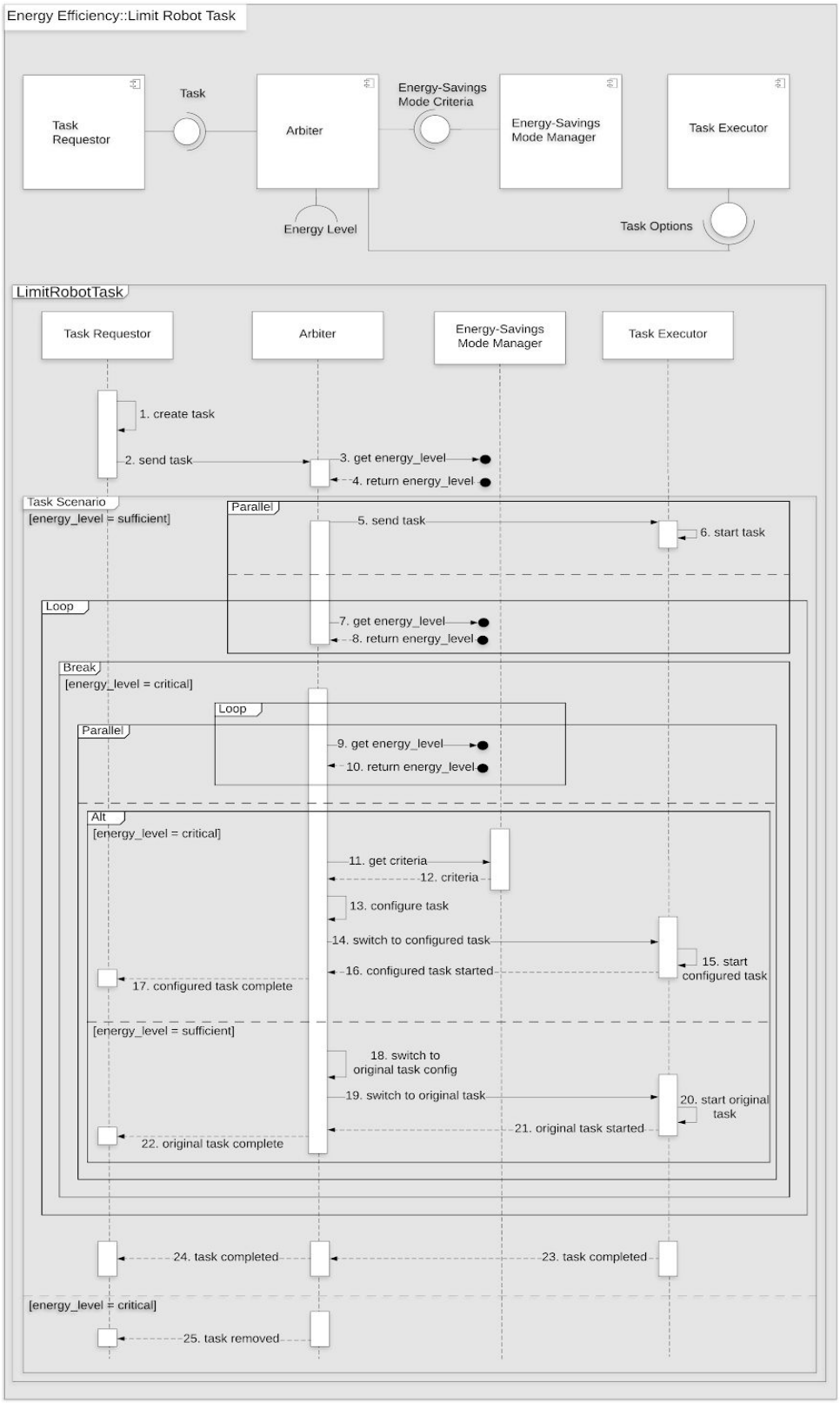
EE1: Limit Task

Tactic Name	EE1: Limit Task
Targeted QA	Energy Efficiency
Family	Energy Efficiency
Motivation	Robotic activities such as data sampling or large amounts of data transfer consume a significant amount of energy. When the robot enters the energy-savings mode, limiting different activities (see Description for list of possible robotic activities) is pivotal in order to meet the energy-savings mode requirements.
Description	This tactic configures a robotic task based on any set criteria for tasks under the energy-savings mode. The *Task Requestor* is responsible for requesting to execute a certain task, the *Arbiter* is responsible for deciding whether or not to configure the original task, the *Energy-Savings Mode Manager* is responsible for providing the criteria for any task under the energy-savings mode (energy-level is critical), and the *Task Executo*r is responsible for executing the original or configured task. First, the *Task Requestor* sends an initial task to the Arbiter (label 2). After receiving the task, the *Arbiter* checks the energy-level of the robot (labels 3,4) provided by another component in the system. If the energy-level is critical, the *Arbiter* immediately removes the task (label 25), thus, not forwarding it to the *Task Executor*. In the case when the energy level is sufficient, in a separate thread the *Arbiter* forwards the task to the Task Executor (label 5) and the Task Executor starts executing the task (label 6). In a parallel thread within a loop, the Arbiter also starts checking the energy-level (labels 7,8). If during the execution of the original task the energy-level becomes critical, the *Arbiter* breaks out of the loop. A new loop is started, and in a separate thread, the *Arbiter* starts checking the energy-level of the robot (labels 9,10). If the energy-level is critical,

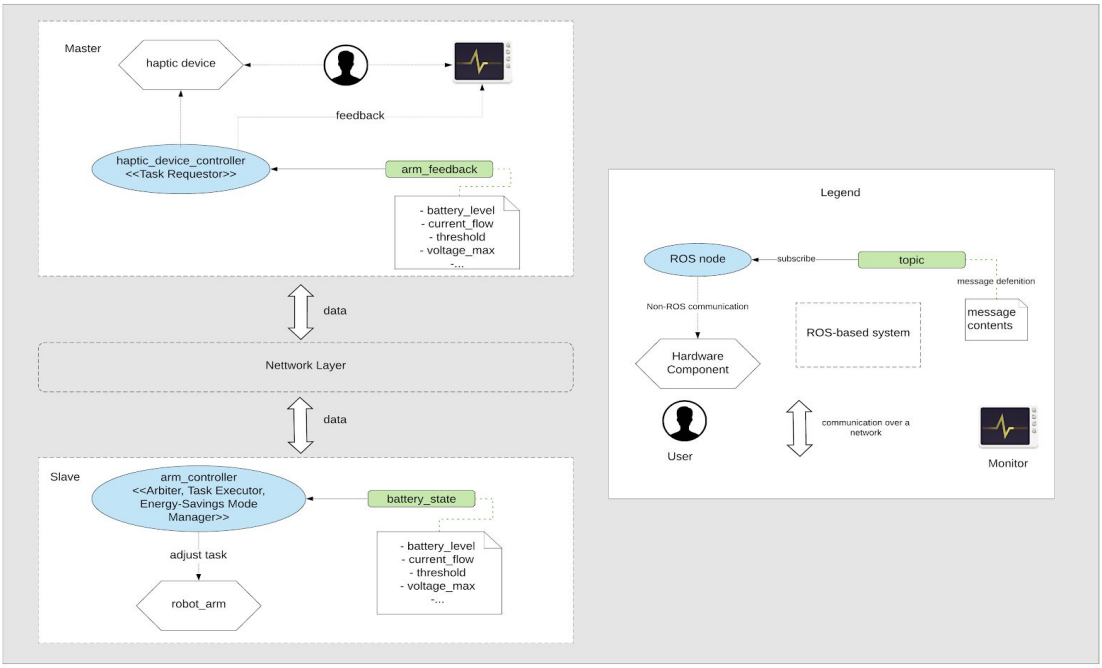
	in a parallel thread, the *Arbiter* gets the energy-savings mode criteria for the task provided by the *Energy-Savings Mode Manager* (labels 11, 12), configures the task (label 13), and instructs the *Task Executor* to start executing the configured task instead of the original task (label 14). If the energy-level becomes sufficient during the execution of the configured task (label 17), the *Arbiter* re-configures the configured task back to the original one (label 18) and instructs the *Task Executor* to start executing the original task instead of the configured one (label 19). If the energy-level drops again to a critical point, the previously described steps will take place (labels 11-15). Some examples of task configurations: ● Initial task: move in any direction at a set max power rate. Configured task: adjust power rate to a lower rate specified in the set criteria ● Initial task: publish large amounts of data (e.g., publish PCL point clouds, 3D map). Configured task: stop publishing data ● Initial task: sample data at a set configurable rate. Configured task: alter the sampling rate to a lower rate specified in the set criteria
Constraints	The tactic as described assumes that the communication between the component providing the energy level information and the *Task Filter Component* already exists, and that the criteria for any task under the energy-savings mode is already set.
ROS Example	Data point #36: Figure 1(b) is an example of how the *Limit Task tactic* can be employed in a ROS-based system with haptic devices. Haptic teleoperation allows a user to perform manipulation tasks in distant, scaled, hazardous, or inaccessible environments. "Haptic teleoperation provides telepresence by allowing a user to remotely control a slave robot through a master device while haptically perceiving the remote environment."[12] In this example, the *Master* represents a ROS-based system which consists of a *haptic_device_controller node* (maps to the *Task Requestor*) which communicates directly with the physical *haptic device*. The *Slave* ROS-based system consists of an *arm_controller_node* (represents the Arbiter, Task Executor and the Energy-Savings Mode Manager) which communicates directly with the *robot arm*. The *arm_controller_node* is subscribed to a battery_state *topic* which is published by some other node. This topic publishes information regarding the robot arm battery such as the battery percentage. The *Slave* ROS-based system communicates the data relevant to the robot arm to the *Master* ROS-based system and vice versa via the network. In the *Master*, the *haptic_device_controller node* is subscribed to an *arm_feedback topic* which publishes messages regarding the state of the robot arm such as the battery percentage. This information is directly communicated by the *haptic_device_controller node* to the user monitor which is used by an actual user controlling the *haptic device*. If the *arm_controller_node* (*Slave*) receives some task from the *Master*, and the robot arm battery level is critical, the *arm_controller_node* adjusts the task according to some criteria for tasks under the energy-savings mode. Simultaneously, the *haptic_device_controller node* in the *Master* is subscribed to the *arm_feedback topic* and communicates the battery feedback to the user monitor. In this way, the user is aware of the robot arm's battery state.
Other Examples	● Cloud integrated sensor network[13]: when energy needs to be saved, data-transmission techniques are replaced with energy-efficient data-transmission techniques such as a customizable sensor information system model which used to modify data transmission and frequency of data collection to make it energy efficient; this approach also reduces CO2 emissions.
Dependencies	This tactic may involve tactics ● EA1 for configuring a task to abort when the energy-savings mode is on ● EA2 for configuring a task to stop and recharge the battery when the energy-savings mode is on ● EA3-EA6 to get the energy level state
Variations	● Event-based logic can be used for getting the energy-level information: for example, a global

[12] https://www.researchgate.net/publication/329952784_Framework_for_Haptic_Teleoperation_of_a_Remote_Robotic_Arm_Device
[13] https://www.hindawi.com/journals/js/2018/1597089/

	value (e.g., ROS topic) can be accessed at any execution point during runtime.



EE1 Tactic



EE1 ROS Example

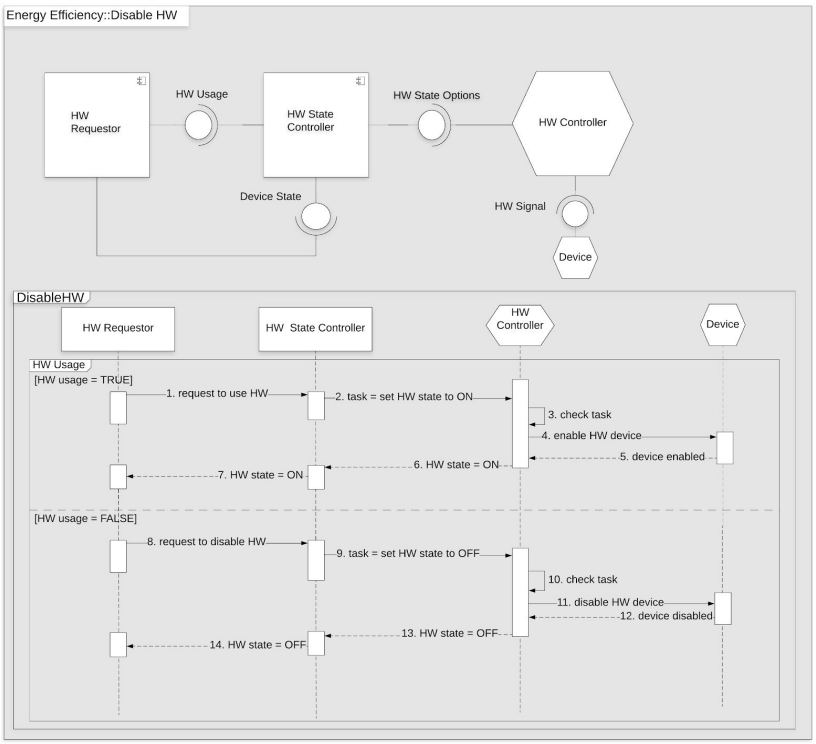
EE2: Disable HW

Tactic Name	EE2: Disable Hardware
Targeted QA	Energy Efficiency
Family	Energy Efficiency
Motivation	Hardware components (e.g., sensors, drivetrains) of a robot often consume a significant amount of energy; for example, sensors consume energy in terms of CPU usage (i.e., sensors provide feedback based on the robot's surroundings and send it to the CPU which then uses the feedback to make further decisions[14]). It is crucial to prevent unnecessary utilization of hardware resources in order to extend the maximum operational time of the robot's battery.
Description	This tactic disables hardware components when they are not strictly needed, which results in the robot consuming less energy and in a more efficient power management during its tasks. The tactic is implemented to control the physical communication between the *HW Controller* and the actual hardware device. The *HW Requestor* notifies the *HW State Controller* whether or not the hardware device is needed for a certain task (labels 1, 8). Then, the *HW State Controller* instructs the *HW Controller* to disable or enable the hardware device (labels 2, 9). Before enabling or disabling the hardware device, the *HW Controller* checks the task to determine if it is safe to change the state of the hardware device (labels 3, 10) (e.g., toggle hardware pin, set boolean variable for instruction to TRUE/FALSE). In this tactic, we assume that it is always safe to either enable or disable the *HW device*, so the *HW Controller* enables or disables the *HW device* (labels 4, 11), based on the input from the *HW State Controller*.

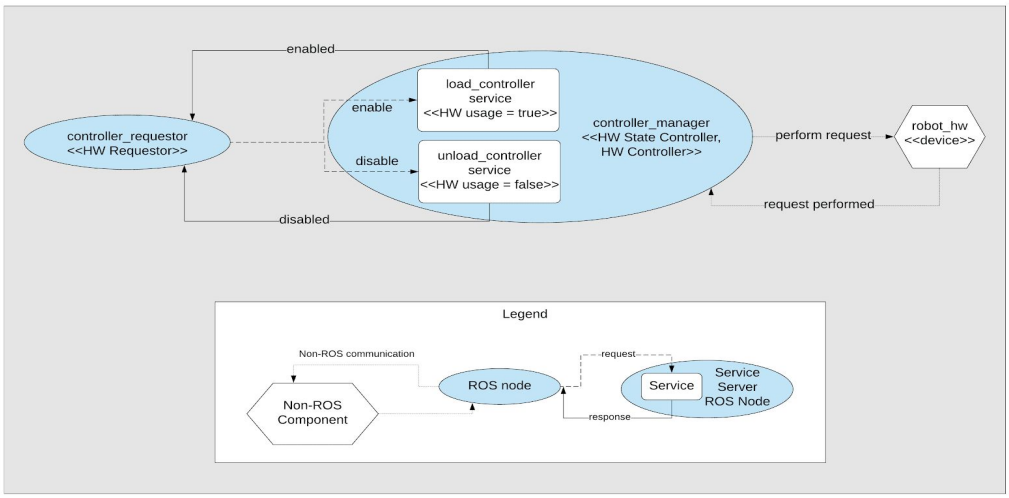
[14] https://link.springer.com/chapter/10.1007/978-1-84628-642-1_5

Constraints	The tactic as described assumes that the communication between the actual hardware device and the *HW Controller* already exists and that it is safe to change the state of a hardware device. It is also assumed that other components in the system already notified the *HW State Controller* regarding the demand for a hardware device.
ROS Example	Data Point #23: Figure 1(b) illustrates an example of the *Disable Hardware tactic* and how it is employed in ros_control package which includes ROS-based controller managers and controller and hardware interfaces. There are two nodes: a *controller_requestor node* (maps to the *HW Requestor*)and a *controller_manager node* (maps to the HW State Controller and HW Controller). There is also one non-ROS component - the *robot_hw* (maps to the *Device)*. The controller_manager advertises two services: a *load_controller service* (enable HW) and an *unload_controller service* (disable HW). If the *controller_requestor node* wishes to enable the *robot_hw*, it sends a request to the *load_controller service* to enable the *robot_hw*. If it wishes to disable the *robot_hw*, it sends a request to the *unload_controller service* to disable the *robot_hw*. After receiving a request, the *controller_manager node* performs the request by either enabling or disabling the *robot_hw*. The *controller_requestor node* is notified with the result; this ensures that the *controller_requestor node* is aware of the *robot_hw* status. Topics are used to communicate battery information to any subscribed ROS node*, as* topics are used for one-way continuous data flows and no feedback from the subscriber node is needed[15]. The communication between the controller and the controller_manager node is ROS-free since it involves physical signals.
Other Examples	• Windows power management - when the computer enters a sleeping state, Windows puts the network interface card to sleep as well. • Plug-in hybrid electric vehicles - disable certain drive modes for a better power management. • Mobile activity recognition system (ARS) for detecting human activities - disable GPS hardware while the user is indoors.[16] • LG air conditioners - power consumption is reduced by automatically turning off the circulation fan as well as the exhaust fan when the compressor is off.
Dependencies	-
Variations	• In the case when switching the state of the hardware device is not safe, the *HW Controller* will check the task and suspend it. • The *HW State Controller* can check whether or not the actual hardware device is being currently utilized.

[15] http://wiki.ros.org/ROS/Patterns/Communication
[16] https://link.springer.com/article/10.1007/s00779-013-0638-2



EE2 Tactic



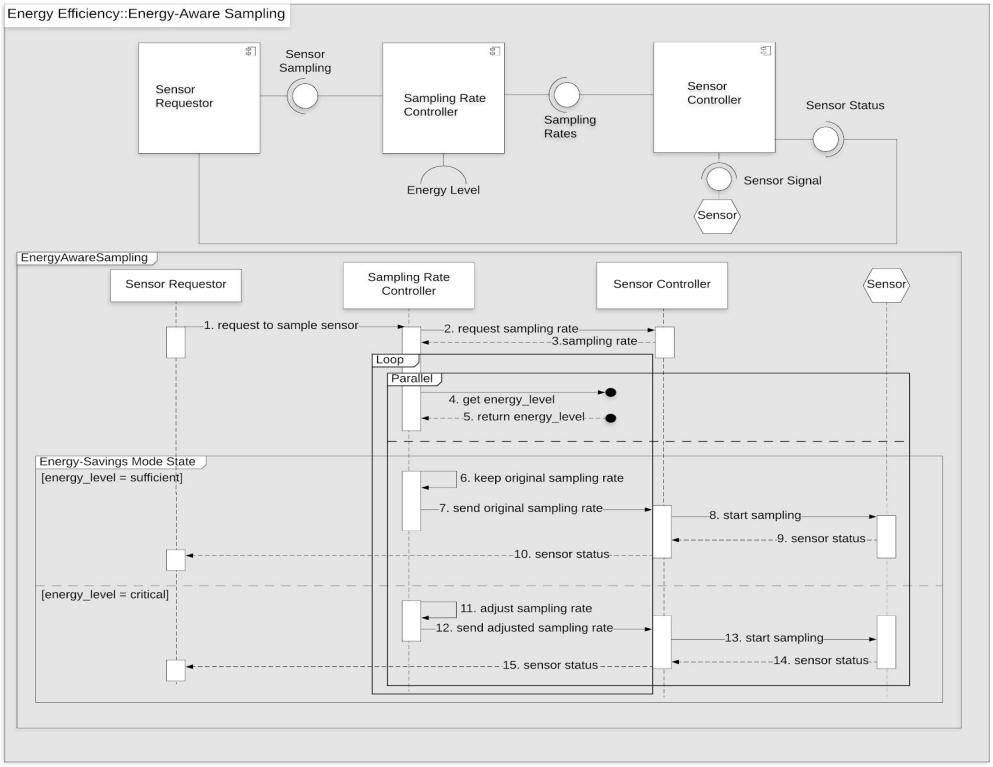
EE2 ROS Example

EE3: Energy-Aware Sampling

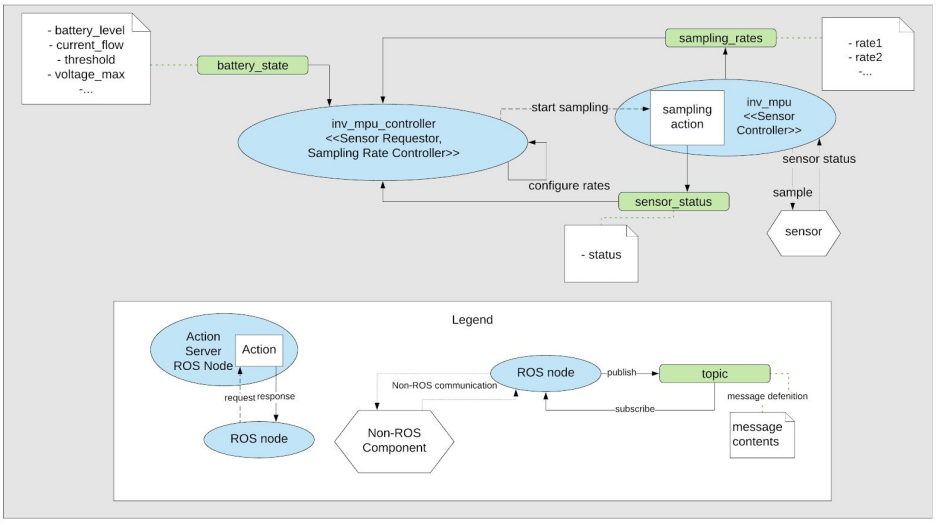
Tactic Name	EE3: Energy-Aware Sampling
Targeted QA	Energy Efficiency
Family	Energy Efficiency

Motivation	In robotics, many sensors are designed to provide a continuous stream of data (e.g., accelerometers, lidars)[17]. Sampling data from sensors is an energy consuming task which is used to analyze and/or control a sensor. When the robot's battery reaches a critical point, the component in charge of sampling the sensor should be able to continue sampling, and at the same time, enter into a state in which energy must be saved to avoid further drain of the battery.
Description	This tactic adjusts the rates for sampling a sensor based on the energy-level of the robot. Figure 1(a) illustrates the component interface and sequence diagrams of the tactic. The *Sensor Controller* provides configurable sampling rates for the sensor. First, the *Sensor Requestor* requests to start sampling the sensor and sends the request to the *Sampling Rate Controller* (label 1). After receiving the request, the *Sampling Rate Controller* gets the initial sampling rate from the *Sensor Controller* (labels 2,3). Then, in an infinite loop and in a separate thread the *Sampling Rate Controller* gets the energy-level which is provided by another component in the system (labels 4,5); if the energy-level is critical (e.g., the battery level is below a set threshold), the initial sampling rate is lowered (within the same loop and in a parallel thread) (label 11). The adjusted sampling rate is then sent back to the *Sensor Controller* (label 12) which uses the adjusted sampling rate to sample the sensor (label 13). The sampled sensor state information is then provided by the *Sensor Controller* which communicates this information back to the *Sensor Requestor* (labels 14, 15). Under normal operation (e.g, energy-level is sufficient to operate, battery level above a set threshold), the initial sampling rate is not altered (label 6-10).
Constraints	This tactic assumes that the communication between the physical sensor and the *Sensor Controller* is already established and that the initial sampling rates are already configured. It also assumes that the communication between the component providing the energy level information and the *Sampling Rate Controller* already exists.
ROS Example	Data Point #51: Figure 1(b) illustrates an example of the *Energy-Aware Sampling tactic* and how it is employed in a ROS-based driver for InvenSense's 3-axis gyroscope. There are two nodes and one non-ROS component involved: the *inv_mpu_controller node* (maps to *Sensor Requestor* and *Sampling Rate Controller*), the *inv_mpu node* (maps to *Sensor Controller*), and the *sensor* component. The *inv_mpu_controller node* is subscribed to a *battery_state topic* to check the battery level. It is also subscribed to a *sampling_rates topic* in which the sampling rates are published by the *inv_mpu node.* After getting a sampling rate, the *inv_mpu_controller node* checks the battery level and either adjusts or keeps the original sampling rate based on the battery level. If the battery level is sufficient, the *inv_mpu_controlle node* keeps the original sampling rate and sends a request to the *sampling action* advertised by the *inv_mpu node* to start sampling the *sensor*. If the battery_level is critical, the *inv_mpu_controller node* first adjusts the sampling rate, and then makes a request to the *sampling action* to start sampling the *sensor*. The *inv_mpu node* samples the sensor via non-ROS communication methods and publishes the sensor status to a *sensor_status topic* which is subscribed to by the *in_mpu_controller node*.
Other Examples	• Sensor network for automated water quality monitoring[18] - if the monitored parameters hardly fluctuate, lower sampling frequency is used; less energy will be needed for data sampling, data processing and transmission. • Mobile activity recognition system (ARS) for detecting human activities[19] - energy-efficient ARS - low sampling rates, can achieve high recognition accuracy and low energy consumption.
Dependencies	This tactic can employ tactics EA3 or EA4 to get the state of the energy level.
Variations	• Event-based logic can be used for checking the energy-level: for example, a global value (e.g., ROS topic) can be accessed at any execution point during runtime.

[17] https://www.springer.com/gp/book/9783540003359
[18] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5713192/
[19] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5621091/



EE3 Tactic

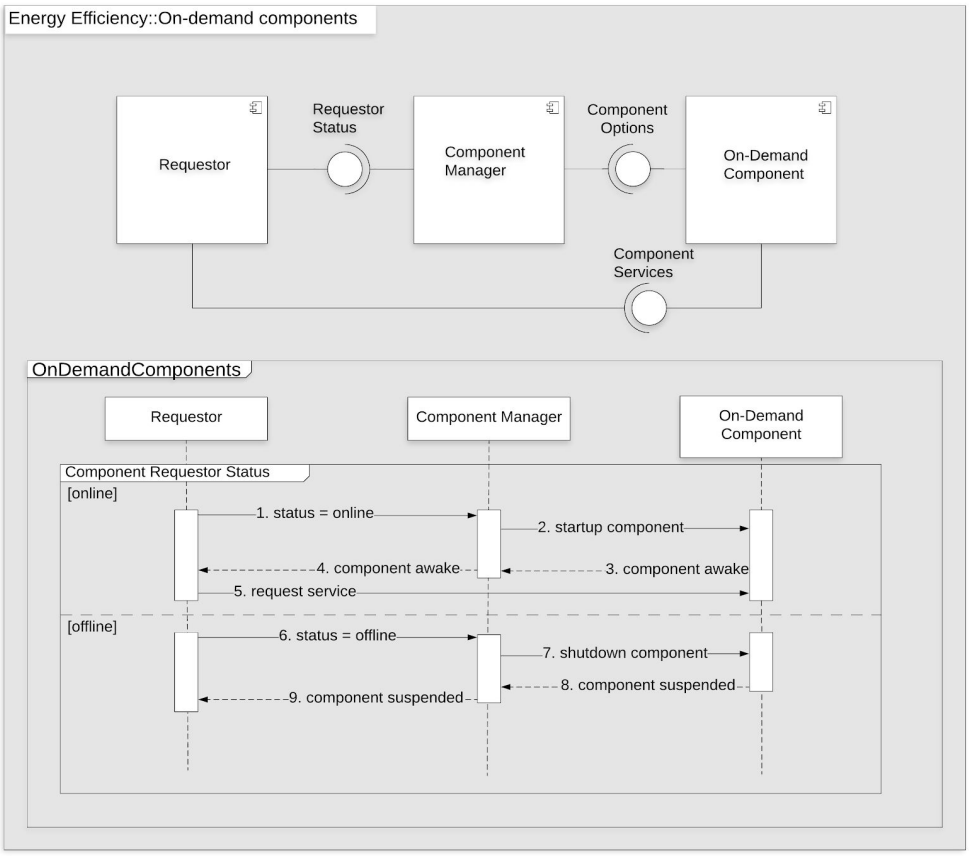


EE3 ROS Example

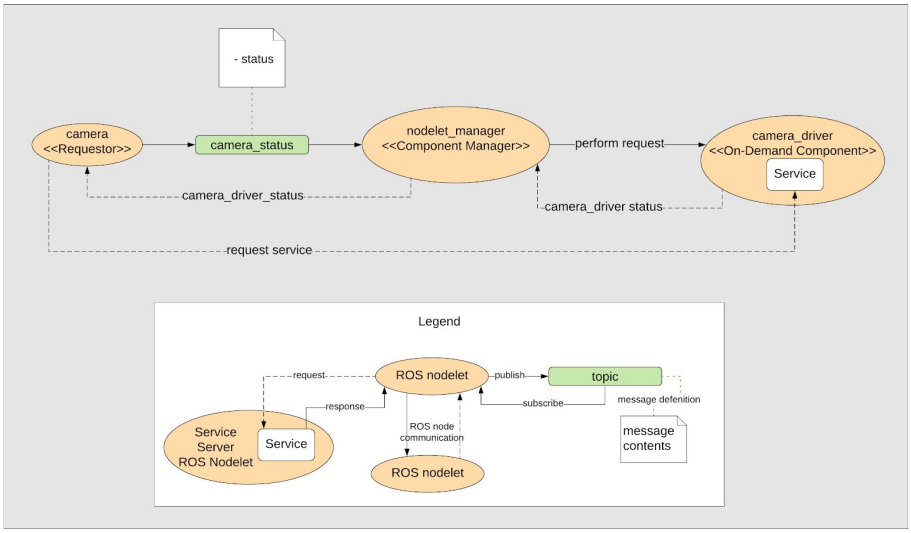
EE4: On-Demand Components

Tactic Name	EE4: On-Demand Components
Targeted QA	Energy Efficiency
Family	Energy Efficiency

Motivation	Continuously running a component requires the spawning of an OS process which is an energy-consuming task in terms of CPU usage (i.e., executing a CPU-intensive loop) and other resources (e.g., sensors, motors, fans for cooling). For this reason, it is necessary to ensure that OS processes that are not being utilized do not consume energy unnecessarily by running in the system.
Description	This tactic brings up new components only when their functionalities are needed. Figure 1(a) illustrates the component interface and sequence diagrams of the tactic. The *Component Manager* acts as a controller which either starts up or shuts down a component based on its demand. The *Requestor* represents a component which requires the *On-Demand Component* based on its status: online (startup a component) or offline (shutdown a component). First, the *Requestor* sends its status to the *Component Manager* (labels 1,6) which either brings up or shuts down the *On-Demand Component* (labels 2,7). The status of the *On-Demand Component* is communicated from the *Component Manager* to the *Requestor* (labels 3,4 & 8,9) so that it is aware whether or not the *On-Demand Component* is up and running; if it is up and running, the *Requestor* can start communicating with the *On-Demand Component* (label 5).
Constraints	It is assumed that *On-Demand Component* is already implemented and is waiting to be brought up or shut down.
ROS Example	Data Point #14: Figure 1(b) is an example of how the *On-Demand ROS Component* tactic can be employed in a practical scenario when dealing with cameras. In this scenario, ROS nodelets are used instead of ROS nodes. The reason behind this is that when ROS nodes are dealing with messages that contain large amounts of data (i.e. camera images, point clouds), sending and unpacking the messages takes a longer time over TCP. With nodelets, messages are communicated *within the same process* via a *booster shared pointe*r. This helps to reduce overhead of data transfer. In this example, there is a *nodelet_manager* (maps to Component Manager) which has a pool of threads shared among the nodelets spawned within the same *nodelet_manager*. The *camera nodelet* (maps to Requestor) is a nodelet that is one of the threads running within the *nodelet_manager*. In order for the camera to operate, it requires the *camera_driver nodelet* (maps to On-Demand Component) to be up and running. The *camera nodelet* publishes its status (online, offline) to a *camera_status topic* which is subscribed to by the *nodelet_manager*. Based on the *camera nodelet's* status, the *nodelet_manager* either starts up or shuts down the *camera_driver*. The *camera_driver nodelet's* status is communicated back to the *camera nodelet*. If the *camera_driver nodelet* is up and running, it advertises a Service which can be used by the *camera nodelet*.
Other Examples	• Cloud computing - delivery of on-demand computing resources, ability to scale computing resources. • iOS - on-demand resources (ODR) API to reduce original download size of the application. Instead, the ODR API is used to deliver the application contents *after* it has been downloaded. This is especially useful for gaming applications.
Dependencies	-
Variations	



EE4 Tactic



EE4 ROS Example