

Team Members : Sindhu Satish and Upsham Dawra

GizmoE System Architecture

Team GizmoE

Carnegie Mellon University

Table of Contents

Purpose of the Document 3

Document Reading Guidelines 3

Project/ Business Context 3

Stakeholders 4

Project Scope 4

Functional Requirements 5

Composition-Time Functional Requirements 5

Execution-Time Functional Requirements 5

Quality Attributes 6

Architecture Description 10

Task Composition Architecture 11

Task Execution Architecture 13

Capability Addition Architecture 14

Illustration of Task Tree Resolution 15

Static View 24

Analysis & Tradeoffs 25

QAS01 Run-time extensibility 25

Task Composition Architecture 25

Task Execution Architecture 26

Capability Addition Architecture 26

QAS02 Static reusability 26

Task Composition Architecture 26

Task Execution Architecture 26

Capability Addition Architecture 27

QAS03 Static Modifiability 27

Task Composition Architecture 27

Task Execution Architecture 27

Capability Addition Architecture 27

QAS04 Reliability 28

Task Composition Architecture 28

Task Execution Architecture 28

Capability Addition Architecture 28

QAS04 Reliability 28

Task Composition Architecture 28

Task Execution Architecture 28

Capability Addition Architecture 29

QAS05 and QAS06 Performance 29

Task Composition Architecture 29

Task Execution Architecture (QAS05) 29

Capability Addition Architecture 30

Appendix 30

# Purpose of the Document

This document highlights the architectural drivers motivated by the Business Context and a detailed description of the system architecture. It also talks about the key design decisions and tradeoffs that were made to satisfy the architectural drivers.

# Document Reading Guidelines

This document has 7 sections; Project/Business Context, Stakeholders, Project Scope, Functional Requirements, Quality Attributes, Architecture Description, Analysis & Tradeoffs. To be able to better understand the motivation behind the system design, it is good to get a through reading of the Business Context, Stakeholders, and the Project Scope. After this is understood, you may go on to read the Functional Requirements (which are categorized as Composition-Time and Execution-Time) and Quality Attribute Scenarios. While reading the document you may find terms like “Capability” “Task” “Composition-Time” etc. to better understand these terms it is suggested that you read the Appendix which has brief descriptions of the terms specific to this project. Note that we have tried to create the architectural diagrams that are self-explanatory with legends, good naming conventions, etc. and some of the most significant information regarding system design and how the system design supports the Functional and Quality requirements of the system can be extracted from these diagrams. Hence, it is strongly suggested not to ignore the architectural diagrams and other supporting illustrations to get a better understanding of the system.

# Project/ Business Context

Teleconferencing facilities allow remote participants to access organized meetings, as long as the meeting is in a space that houses teleconferencing equipment. A Tele-Presence robot (CoBot) improves this capability by making teleconferencing capabilities mobile. Before our project, much work has been done on the CoBot and it is capable of understanding “tasks” or CoBot capabilities. However, there is no good way for anyone to really use the CoBot System; the current user interface is very primitive. We will be aiming to make this system more usable. In specific, there is a need to allow the users to easily compose and execute tasks. In addition, the capability developers should be able to add and delete capabilities in the system.



Fig. 1. System Boundary

# Stakeholders

The stakeholders of project GizmoE are Sponsors, the MSIT Development team, the Robotics Team and the remote users . The Sponsors/Clients are Dr. Bradley Schmerl and Dr. David Garlan of the School of Computer Science, Carnegie Mellon University. They are researchers who are interested in seeing the implementation of their vision of a highly extendible, plug-n-play like architecture of a system that allows easy composition and execution of tasks by naïve remote users through a robot like CoBot.

The Development Team comprises of two students, Upsham Dawra and Sindhu Satish from the MSIT-SE program from the Institute of Software Research, Carnegie Mellon University. This team has been assigned to fulfill the requirements of the project sponsors.

The Robotics Team, who created the CoBot is interested to see how far can the Sponsors and their Development team can get in using the CoBot and its features to allow tele-conferencing.

The remote users are those who are interested in gaining access to a remote space where the CoBot is available for use and perform meaningful tasks like attending a poster session, meeting, etc.

# Project Scope

The aim is to design a sustainable and extendable architecture of a system that will allow a user to compose tasks in a simple and interactive fashion. Another aim is to enable developers to easily build and deploy new capabilities on the system. Once the system is designed and validated, we will look to provide a prototype (subject to negotiation with the clients) to demonstrate the robustness of the architecture. It is important to stress the fact that we will not be adding direct functionality to the CoBot (for instance, new capabilities), but will be building a framework to actually use the CoBot. We will be working under the assumption that the CoBot can be used for a maximum of 4 hours before it needs to charge for 1 hour.

# Functional Requirements

The functional requirements for this system can be broadly categorized into Composition-Time requirements and Execution-Time (Run-time) requirements for a clear understanding of the scope of these requirements.

## Composition-Time Functional Requirements

1. Task composition

* The user should be able to compose tasks out of existing capabilities and tasks.
* The use should be able to compose tasks that can be executed in sequence or in parallel.
* The user should be able to provide necessary inputs to the either capabilities or tasks when he is done composing and wishes to execute them.
* The user should be able to provide inputs to the capabilities or tasks at any (reasonable) point during the course of the task execution.
* The user should be able to save the tasks he has composed for future use.
* The user should be able to retrieve the pre-composed tasks for execution.

1. Type-Checking

* There should be a mechanism, which allows only “meaningful” tasks to be composed in sequence, parallel or both.
* There should be a mechanism, which ensures that the user provides “meaningful” inputs to the tasks or capabilities.

## Execution-Time Functional Requirements

1. Interrupt-based task execution

* The status of a task’s execution should be made available for monitoring.
* The user should be able to interrupt/ kill/ resume the execution of tasks at any point in time.
* There should be a mechanism through which the parent tasks are aware of completion of their child tasks.

1. Error handling

* There should be a mechanism, which ensures the inputs and outputs from one capability/task are successfully collected for future retrieval or sent forth to other capabilities/tasks below it.
* There should be a mechanism, which gives the user the power to take an appropriate action when an error occurs.
* There should be a mechanism that constantly monitors the battery consumption of the CoBot and informs the user when it is below a certain threshold where the user will have to abort tasks that are executing and command the CoBot to navigate to the charging station.

1. Addition of new capabilities

* The developer should be able to easily use the existing framework (designed by team GizmoE) to build new capabilities that can be deployed for use on the system.

# Quality Attributes

Considering the above functional requirements that focuses a great deal on the level of interactivity between the user and the system and the fact that this system involves the use of a robot, following are some key quality attributes:

1. **Run-time extensibility:** The developers should be able to use the framework to develop newer capabilities and add them to the system at run-time. These new capabilities will be available for task composition to the user without any disruption in his session of system/ CoBot use.

Following is a quality attribute scenario that depicts the same:

|  |  |
| --- | --- |
| Scenario Title: Runtime Extensibility Scenario 1 | Scenario ID: QAS01 |
| Raw Quality Attribute Description: A capability developer for the system wants to develop a new capability and add to the existing system. This new capability should be added into the capabilities repository for the user to be able to use it for task composition, with minimal changes to the existing system and minimal effort. | |
| Source of Stimulus: | Capability developer |
| Stimulus: | Need for additional capabilities |
| Environmental Condition(s): | The user is in the middle of his session of using the system/ CoBot to compose and execute tasks |
| System Element(s) (artifact): | Database, GUI elements (buttons, dropdown menus etc), and the code modules that will be developed for this new capability and those with which it integrates/ interfaces with |
| System Response: | Change in GUI and database |
| Significant Measures: | * Developer effort for appending new capability data into database table should be less than 2 man hours * Developer effort for adding new GUI elements (buttons, drop down menus) should be less than 2 man hours * Developer effort for developing the code modules should be less than 300 lines of code * % of changes that need to be made to the existing system (code) while adding new capability should be less than 20% * % of code reused should be greater than 30% |

1. **Static reusability:** The developers or extenders of this system should be able to re-use the existing framework to in a different context or for system extension.

Following is a quality attribute scenario that depicts the same:

|  |  |
| --- | --- |
| Scenario Title: Static reusability | Scenario ID: QAS02 |
| Raw Quality Attribute Description: A developer of the system should be able to re-use the existing code modules via the framework provided for extending the system or for use in a similar context. | |
| Source of Stimulus: | Developer |
| Stimulus: | The developer wants to extend the system or use the code modules in another context |
| Environmental Condition(s): | The current system code modules are as is |
| System Element(s) (artifact): | Database, GUI elements (buttons, dropdown menus etc.), and the code modules (interfaces and wrapper classes, etc.) |
| System Response: | The system framework allows the developer to extend the system by using the same code |
| Significant Measures: | * % of code reused from the current system should be greater than 30% while extending |

|  |  |
| --- | --- |
| Scenario Title: Static Modifiability | Scenario ID: QAS03 |
| Raw Quality Attribute Description: A developer of the system should be able to re-use the existing code modules via the framework provided for modifying the system to add new functionality | |
| Source of Stimulus: | Developer |
| Stimulus: | The developer wants to extend the system or use the code modules in another context to add new functionality |
| Environmental Condition(s): | The current system code modules are as is |
| System Element(s) (artifact): | Database, GUI elements (buttons, dropdown menus etc.), and the code modules (interfaces and wrapper classes, etc.) |
| System Response: | The system framework allows the developer to extend the system by modifying the code |
| Significant Measures: | * % of code modified from the current system should be not be greater than 30% while adding new functionality |

1. **Reliability:** The user should be at the least made aware of a failed task, even if the system is unable to correct it. Although there is a deep interest in exploring various ways of handling reliability (like fail silently, rollback, etc which will be dealt with in the architectural tactics segment of the main document), ensuring some form of reliability is important. In addition, it is important to not loose control of the CoBot when there is a disruption in connectivity.

Following are the quality attribute scenarios depicting the same:

|  |  |
| --- | --- |
| Scenario Title: Reliability Scenario 1 | Scenario ID: QAS03 |
| Raw Quality Attribute Description: Whenever a connection with the Cobot is lost, system should detect it and report it to user. | |
| Source of Stimulus: | Network connection |
| Stimulus: | The system loses connection with the CoBot |
| Environmental Condition(s): | The system is running |
| System Element(s) (artifact): | The system elements (client-side application, the server, the CoBot) |
| System Response: | The server listens for the heartbeat from the developer added capabilities to ensure that the connection is not lost. |
| Significant Measures: | * A failure with the connection to the CoBot should be notified to the user within 15 seconds |

|  |  |
| --- | --- |
| Scenario Title: Reliability Scenario 2 | Scenario ID: QAS04 |
| Raw Quality Attribute Description: There should be a reliable communication through events between the user and the capabilities. | |
| Source of Stimulus: | User |
| Stimulus: | Interrupt message to stop the execution of capabilities |
| Environmental Condition(s): | The capability is executing normally |
| System Element(s) (artifact): | Client-side application, the server, the CoBot, the capability |
| System Response: | On user’s input, the system is able to aborts the current execution of tasks. |
| Significant Measures: | * The capability responds to user’s event within 15 seconds |

1. **Performance:** When tasks are composed and performed in various ways (sequence, parallel or both) and tasks are composed of multiple capabilities, the users should not be kept waiting for feedback from the system for a very long period (during both composition and execution). Hence, it is required that the users face as low a period of latency as possible.

Following is a quality attribute scenario depicting the same:

|  |  |
| --- | --- |
| Scenario Title: Performance Scenario 1 | Scenario ID: QAS05 |
| Raw Quality Attribute Description: The user of the system has started the execution of his composed tasks and experiences delay in its execution. | |
| Source of Stimulus: | User |
| Stimulus: | The user tries to execute tasks/ capabilities |
| Environmental Condition(s): | The system is running normally but communication channel is experiencing high load. |
| System Element(s) (artifact): | client-side application, the server |
| System Response: | The user receives the status of the task execution |
| Significant Measures: | * Latency should be less than 30 seconds |

|  |  |
| --- | --- |
| Scenario Title: Performance Scenario 2 | Scenario ID: QAS06 |
| Raw Quality Attribute Description: The user experiences a latency while retrieving tasks/capabilities for composition | |
| Source of Stimulus: | User |
| Stimulus: | The user tries to view the list of all the predefined tasks |
| Environmental Condition(s): | The system is running normally |
| System Element(s) (artifact): | client-side application, task database |
| System Response: | The system should display the list of available tasks |
| Significant Measures: | * Latency should be less than 20 seconds |

# Architecture Description

The architecture of the GizmoE system can be viewed based on three dominant features utilities, namely:

* **Task Composition:** This view will comprise of the all the architectural components responsible for allowing a user to compose “meaningful” (this will be explained in detail in the later sections of this document) tasks out of other tasks and capabilities.
* **Task Execution:** This view will comprise of all the architectural components responsible for execution of the tasks composed by the user.
* **Capability Addition:** This view will comprise of all the architectural components responsible for allowing a capability developer to add new capabilities to the system that can be used by the users at runtime for composing tasks.

## Task Composition Architecture

Task composition part of the GizmoE system is based on the standard Model-View-Controller design pattern. This pattern has been chosen to ensure separation of concerns and allow easy and simple user interaction. Following are the major architectural components and their description.

Note: The following descriptions of the architectural components highlights the responsibilities allocated to them and the possible ways they all interact with each other. It does not entail reasons behind such responsibility allocation and communication mechanism. These will be dealt with in detail in the section Architectural Decisions.

**Composition View:** As mentioned before, since we are following an MVC approach, this component is a JSP page that will be responsible for interacting with the user who wants to compose tasks. It can be considered as the client-side of the application. It will be responsible for taking and sending the form based inputs from the user for processing. This will server as an interface for the users to interact with the system while executing the tasks he/she composed (ex: Providing inputs during run-time, see functional requirements)

**Composition Controller:** This component is a thread that is responsible for taking the inputs received from the task composition view and sending them to the other components (Rule Module) for processing. It is also responsible for receiving the processed data from those components and commanding the task composition view to update its view accordingly. One of its main responsibilities is to invoke the Task Tree Resolver component. When the user sends some inputs via the task composition view, this controller sends it to the Rule Module to ensure that the inputs are correct (these inputs can be the different tasks or capabilities the user is trying to compose together as one task or inputs to the specific tasks, like room number for Go To Room capability). Once it receives the “Go” from the Rule Module, the controller sends the information (Tasks and capabilities that need to be composed) to the Task Tree Resolver, which generates an execution tree.

**Rule Module:** This is component is an object that is responsible for checking if the tasks and capabilities that the user has composed make up a meaningful composition. It works like a rules engine where in it has algorithms that check for the correctness while chaining together different tasks and capabilities. For example, if a user has tried to compose Go To Room A and Go To Room B at time 9:00AM as a single task, the Type Checker will have the ability to identify that such composition is not possible and sends an error message back to the Task Composition Controller to let the user know that he is composing the wrong set of capabilities. Type Checker accesses the Tasks DB and Capability DB through the Task Accessor/Writer (DAO) to perform this kind of type checking (you will see what information it accesses from the Tasks DB when we explain the responsibilities Tasks DB).

**Task Accessor/Writer:** This component is an object that is called and returned by the Rule Module and responsible for handling the connection between the Rule Module and the Tasks DB. Any request for data available in the Tasks DB is sent to this component, which transforms these requests into database queries and opens up a JDBC connection and queries the database (capable of performing all CRUD operations). Upon retrieval of raw data from the Tasks DB, it transforms this data into a form that is useful and process able by the Rule Module and sends it to it.

**Task Accessor:** This component is an object that is called and returned by the Task Tree Resolver and responsible for accessing the Tasks DB, upon receiving request from the Task Tree Resolver. It only is capable of performing read operation on the Tasks DB, in other words it acts like a Data Access object.

**Capability Accessor/Locker:** This component is an object that is called and returned by the Rule Module and responsible for performing read and update operations on the Capability DB. It performs read when the Rule Module is trying to type-check the meaningful composition of tasks or inputs for the capabilities. It performs update to lock a capability from deletion (during run-time) when that particular capability has been used for composing a task, which is saved in the Tasks DB. In other words, this component is also a form of Data Access Object.

**Tasks DB:** This component is a wrapper over the database that contains the paths to the XMLs of the tasks that the user has already composed. These are stored so that they can also be used to re-compose other tasks with these as composite tasks within another task. This database should have some metadata for the tasks, which would allow fast re-composability. This would include the overall inputs and outputs of the task.

**Capability DB:** This component is a wrapper over the database that contains the paths to the executable java classes of all the capabilities currently installed in the system as well as the their corresponding string names. It also consists of paths to the configuration files that have necessary input and output requirements of each capability.

**Task Tree Resolver:** This component is an object that is called and returned by the Composition Controller and responsible for generating an execution tree of the task composed. This execution tree is traversed to execute the tasks/capabilities accordingly. Following is a depiction of how these trees are traversed during execution. The Task Tree resolver comes in to picture when the user is finally ready to execute the task he/she has composed

## Task Execution Architecture

The task execution part of the system also follows a MVC pattern. Following are the major architectural components and their description.

**Execution View:** This component is a JSP page and is responsible for allowing user to interact with the system during task execution. It will be responsible for displaying the status of execution to the user.

**Execution Controller:** This component is threads, which is a controller for Execution View and is responsible for receiving inputs from the user like, abort the task, end the task, pause the task etc. This kind of interaction is allowed when there are errors in the task execution or when the user simply wants to interact with the task intentionally. It is also responsible for receiving the execution tree from the Task Tree Resolver and sending it forth to the Task Executor for execution. The Task Tree Resolver performs a standard MVC Controller redirect to the Execution Controller, through a simple method call and sends the tree as a parameter to the Execution Controller. The Execution Controller then invokes the Task Executor to execute the task.

**Task Executor:** This component is a thread that is invoked by the Execution Controller and responsible for traversing the execution tree calling the Capability Invoker object.

**Capability Invoker:** This component is an object that is called and returned by the Task Executor and is responsible for performing read and update operations on the Capability DB. It basically retrieves the path to the capability executable(s) and invokes Capability Thread(s) for execution. Besides this, capability invoker also has a pool of threads that can be reused as different capabilities threads. The idea is to minimize the overhead of creating and destroying threads for each capability execution. Thread pool has different set of threads to support the execution of different capabilities although these different thread classes can be derived from the same base class.

**Capability Thread:** This component is a thread that is invoked by the Capability Invoker and it is single unit of execution (i.e. it is a most basic/primitive capability that can be executed by the system). Each of these capability threads has an associated thread id issued by the capability thread pool that is part of the Capability Invoker.

## Capability Addition Architecture

The capability addition part of the system also follows a MVC pattern. A capability developer uses this part of the system to easily add new capabilities to the system. Following are the major architectural components and their description.

**Capability-Add View:** This component is a JSP page that is responsible for allowing a capability developer to interact with the system while adding the newly developed capability.

**Capability-Add Controller:** This component is a controller for the Capability –Add View and is responsible for receiving commands from the Capability-Add view and sending it forth to the Capability Accessor/Writer.

Capability Accessor/Writer: This component is an object that is called and returned by the Capability-Add Controller and is responsible for performing the CRUD operations necessary for adding and deleting capabilities.

**Event Communication Channel:** This component is a process that acts like an event bus. It allows communication between the Execution Controller, Task Executor, and Capability Thread(s). This process is used to let the Execution Controller know that the task is executing a particular capability or that the task has completed execution. This is done by the Task Executor; by sending events to the Event Communication Channel that is read by the Execution Controller. The Capability Thread sends an event to the Event Communication Channel about its status, which is read by the Task Executor to call the Capability Invoker that in turn, invokes the Capability Thread.

Following is the dynamic view of the GizmoE system:

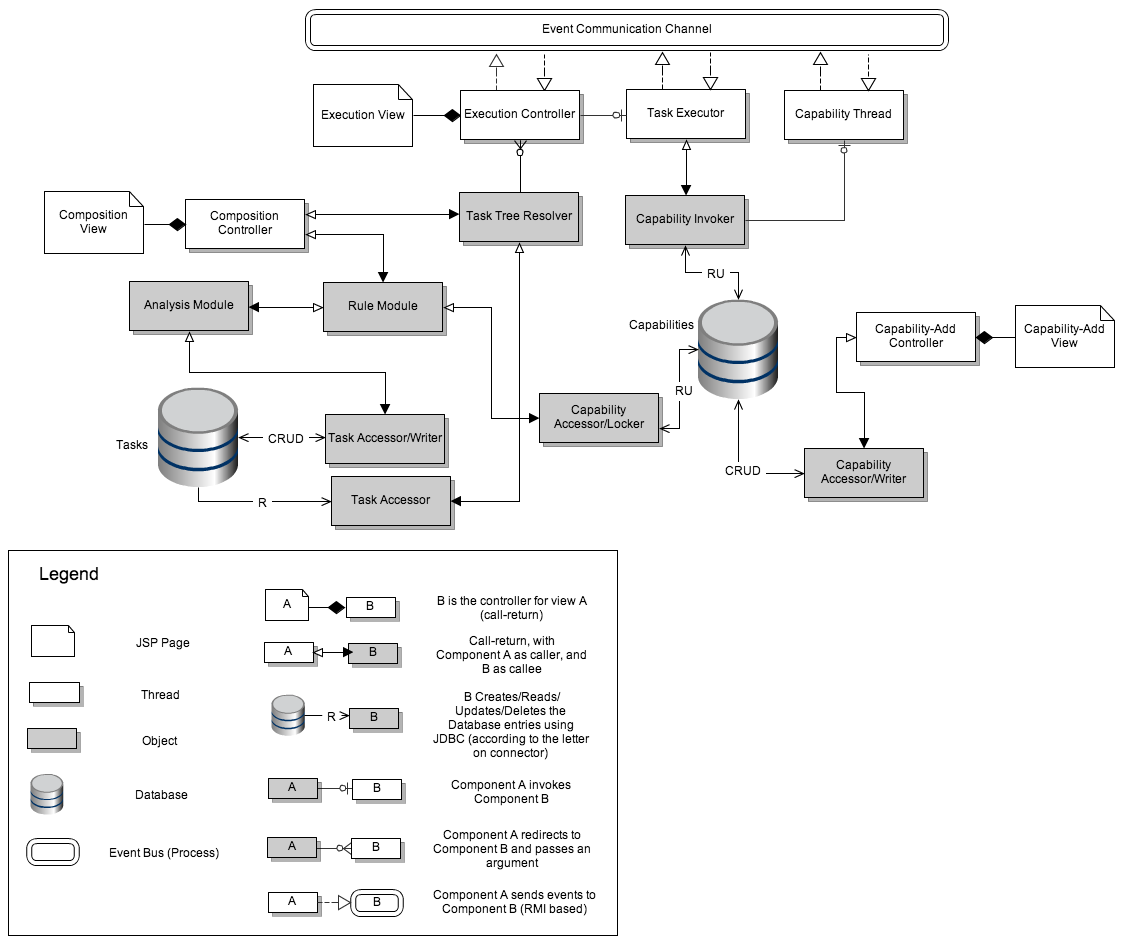


Fig. 2. Dynamic view of the GizmoE system

## Illustration of Task Tree Resolution

The following illustrates a sample execution of a task that comprises of tasks and capabilities in sequence and parallel. This kind of a complex example is chosen to describe all possible execution types (Sequential tasks, parallel tasks, sequential capabilities and parallel capabilities) for a composed task.

The figure below depicts a task which is composed of 4 tasks; Task A, Task B, Task C, Task D. Task A consists of two capabilities in sequence, Task B consists of 3 capabilities in sequence Task C consists of 3 capabilities in parallel and Task D consists of Task B and Task C (Note that a task can be composed of other tasks). This can be visualized as a task upon composition.

## 

Fig. 3. Visual illustration of a complex task composed by a user

Once the task has been composed and ready for execution one can visualize the task execution tree to look like the figure below. This figure shows how each task block has now been resolved into capabilities (which are the most basic units of execution). Initially all the capabilities are in the “to be executed” state. This tree can be considered as an abstraction or visualization of the tree that the Task Tree Resolver generates and sends to the Task Executor. The Task Executor calls the Capability Invoker, which in turn invokes Capability Thread for the first capability in the tree (the root) that is executed.

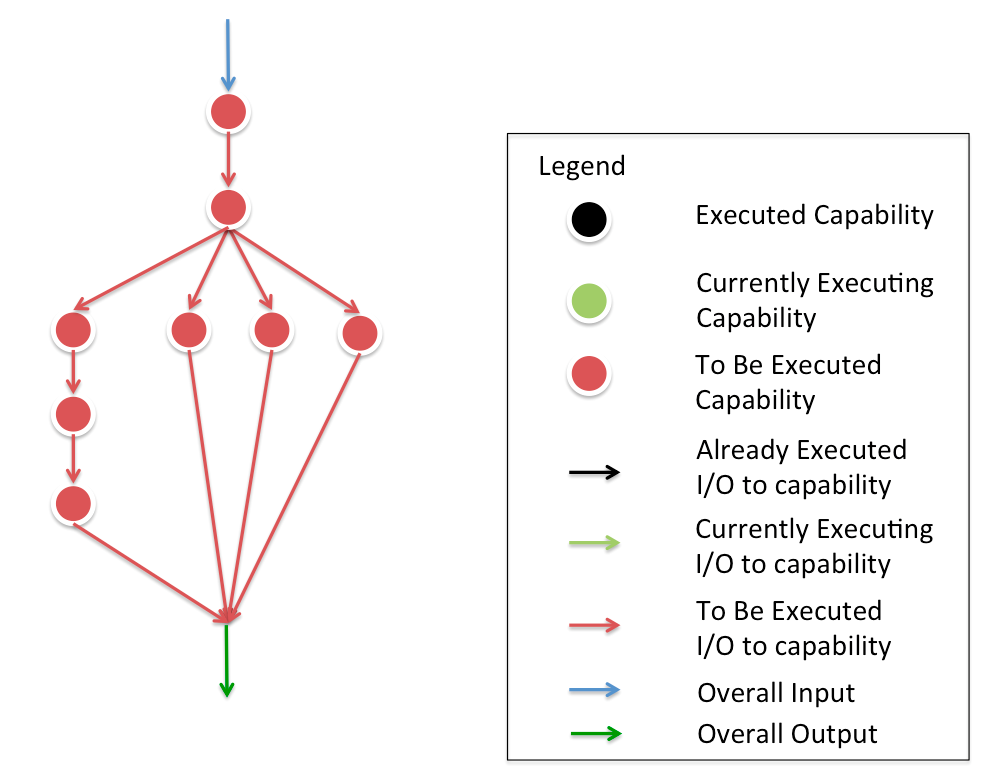


Fig. 4. Visual Illustration of an execution tree of resolved capabilities

Once the first capability is executed, the next in sequence Capability Thread is invoked by the Capability Invoker, which goes into “currently executing” state. All the other capabilities are still in “to be executed” state.

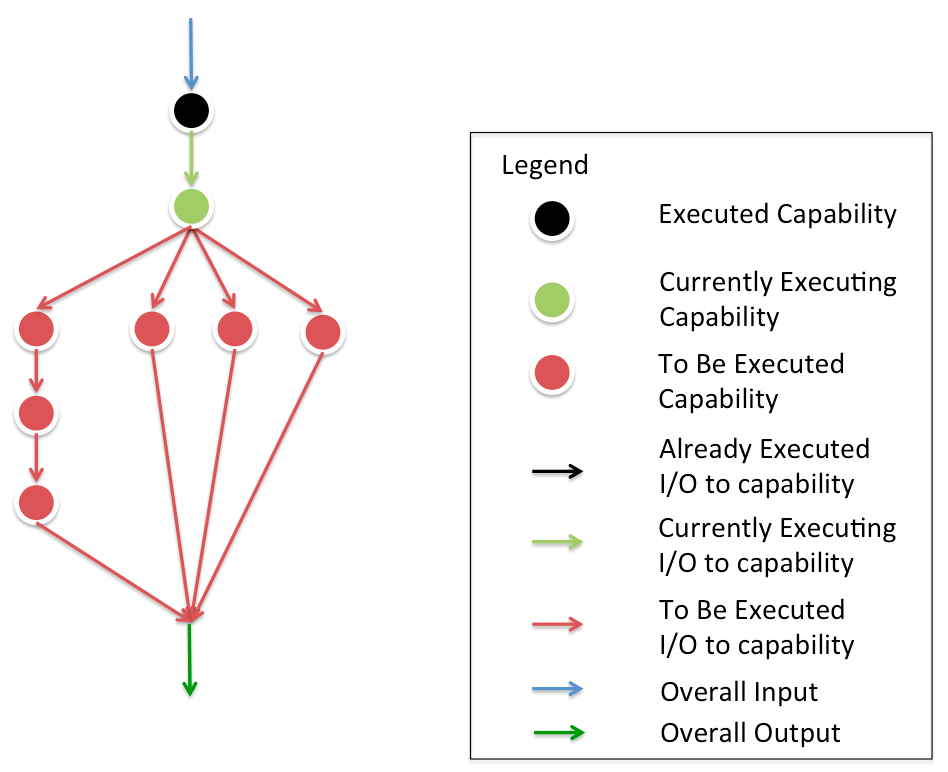


Fig. 5. Visual Illustration of how capabilities are executed in the execution tree

Once the second capability is executed, the four capabilities that is in parallel are invoked and go into the “currently executing” state.

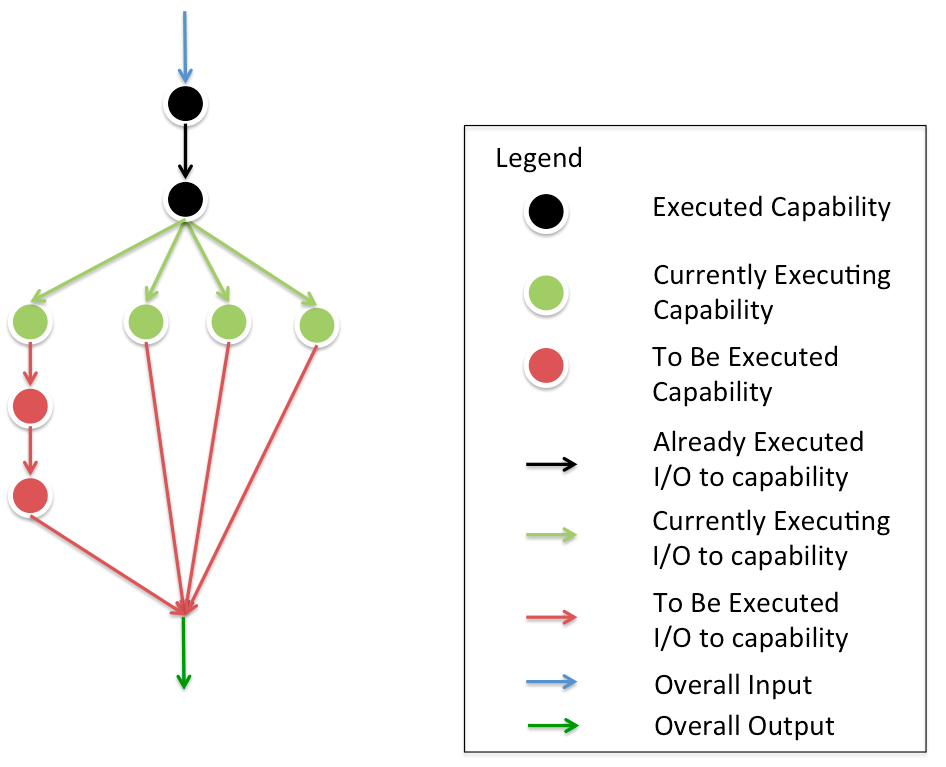


Fig. 6. Visual Illustration of how parallel capabilities are executed in the execution tree

Observer how the overall input is passed to the first capability and how the overall output is finally collected upon completion of all the capabilities.

Following are the illustrations that show the same composition and execution of a task as above reflected in the system architecture:

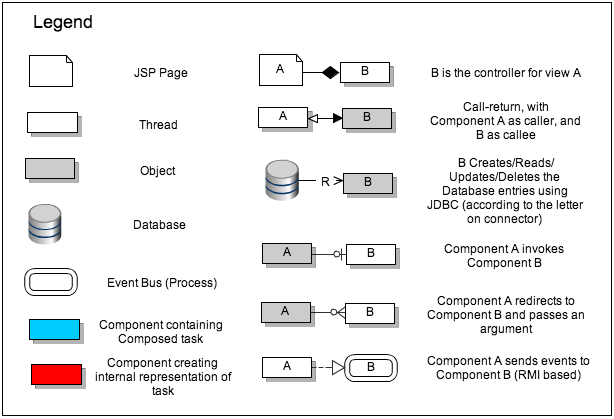


Fig. 7. Legend for figure 8 and 9

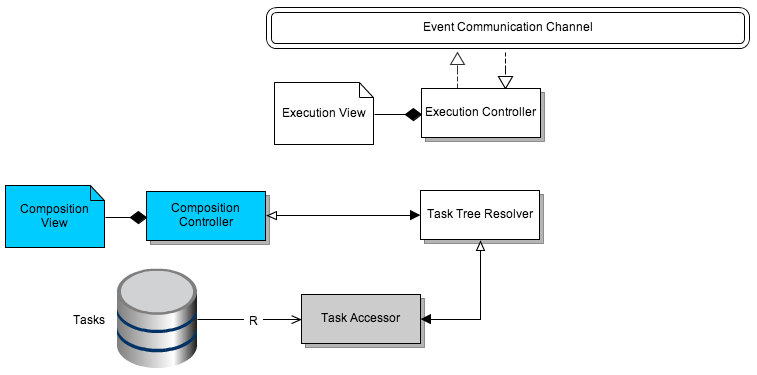


Fig. 8. Illustration of components involved in task composition in the system architecture

The blue components have an internal representation of the composed task. Since the directed a cyclical graph of the capabilities within the composed task has not yet been formed (which is done by the Task Tree Resolver) there is no connection between the Task Tree resolver and Execution Controller in the above illustration.

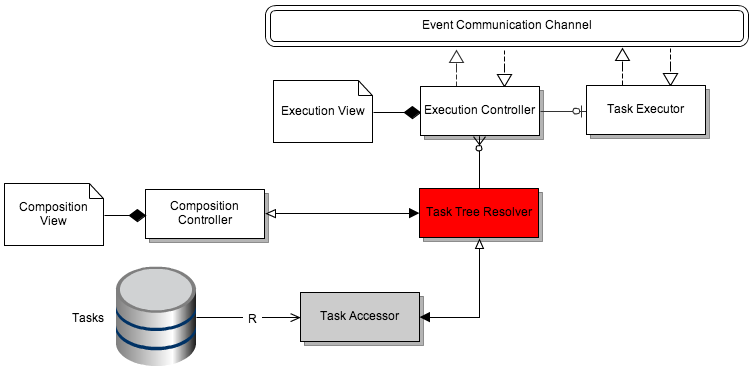


Fig. 9. Illustration of components involved in task tree resolution in the system architecture

The internal representation of a task is resolved in the Task Tree resolver, where we have decided to flatten out the directed acyclic graph of capabilities. This directed acyclic graph is sent to the Execution Controller that spawns a Task Executor for execution of the capabilities.

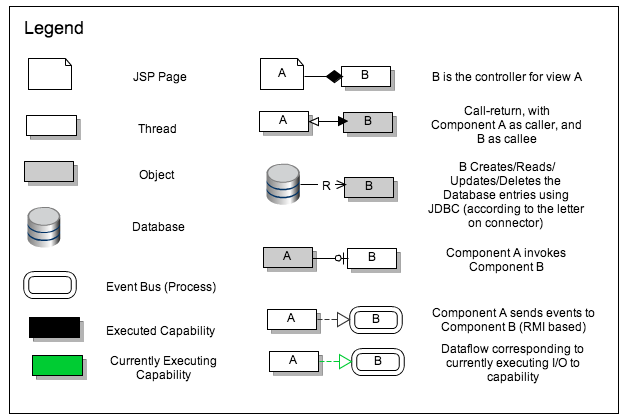


Fig. 10. Legend for figures 11, 12, 13, 14

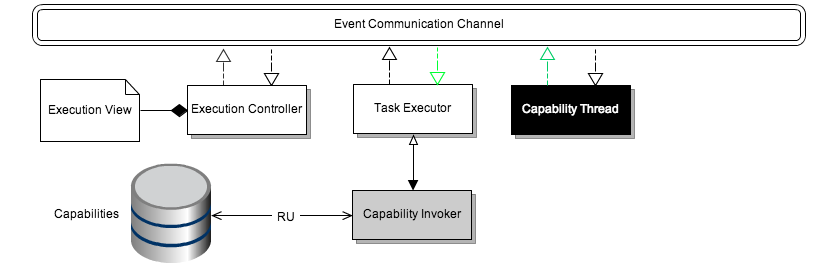
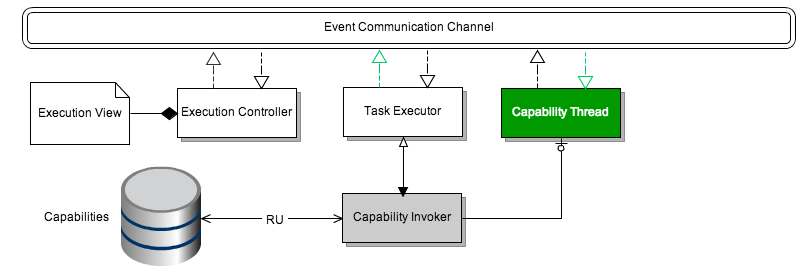


Fig. 11. Illustration of components involved in task execution in the system architecture

In the illustration above you can see that the the first capability in the directed acyclic graph has been executed and it sends an event to the bus which is listened to by the Task Executor (this way the Task Executor knows that a particular capability has been executed and hence can communicate with the user through the Execution Controller).

Fig. 12. Illustration of currently executing thread involved in task execution in the system architecture

In the above illustration you can see how a new capability thread that is part of the generated directed acyclic graph is spawned by the capability Invoker. This capability receives events from the Task Executor with input information.

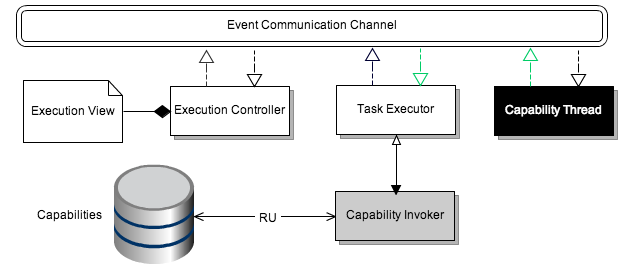


Fig. 13. Illustration of an executed capability thread involved in task execution in the system architecture

Finally, once that capability has been executed, it sends an event that it has completed execution that the Task Executor listens for. This way the user receives status of a capability through the Execution Controller.

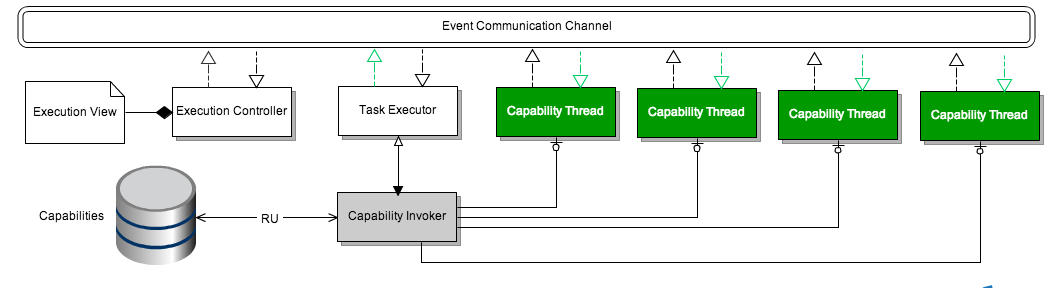


Fig. 14. Illustration of currently executing parallel capability threads involved in task execution in the system architecture

The above illustration shows how the four parallel capability threads are spawned by the Capability Invoker as it is in line to be executed (see figure 6).

Additionally, for error handling, we intend to have a pre-made error handling capability that will be put in parallel with any task. At this point, this is intended to be static capability that will always be added, much like standard error in Linux. However, it is envisioned that such a capability could be added according to context, and could be tailored by the developer too.

## Static View

Following is the static view of the GizmoE system:

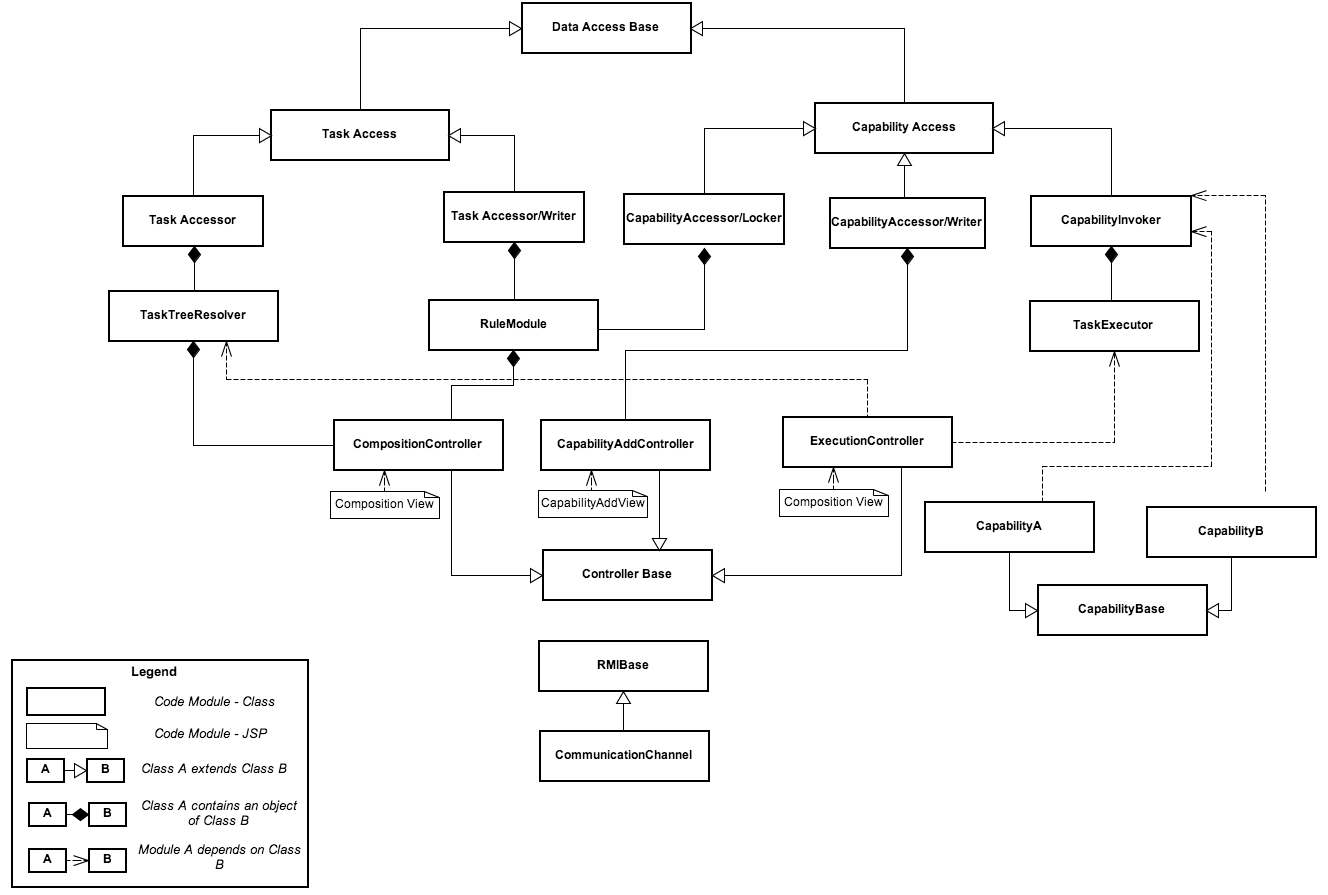


Fig. 15. Static view of the GizmoE system

The static view of the diagram is given below. We have tried to make a framework type of architecture, where we derive classes from base classes. This helps by providing 'presets' when a developer tried to add a capability. For example, when a new developer tries to add a capability, the capability base class will provide a preset of all the required APIs, thus promoting our static reusability concern. This, coupled with a view to add the actual capability in the system, should make adding a capability much easier.

# Analysis & Tradeoffs

## **QAS01 Run-time extensibility**

Here are the key design decisions that support the quality attribute QAS01:

## Task Composition Architecture

* Separate Capability Database with separate administrative interface: Task composition architecture can supports run-time extensibility (QAS01) because capability database and it’s administrative interface is separate from the task composition architecture. During composition of a task, type-checker just look at the current state of capability database. This means that if new capability is reflected in the database then type-checker will take it into the consideration.
* Model View Controller pattern: As already discussed in section 4, task composition interface is dynamic. Since task composition interface is implemented as model-view-controller pattern, it is easier to modify the GUI (view), which may be required by a newly added capability. However it does not support 100% runtime extensibility because sometimes code change may be required on model, view and controller side.
* Locking of used capability: Once a capability is used in a task, it is locked in the capability database and cannot be deleted. This lock is released if one of the following things happen to the task using the capability:
  + Task is cancelled.
  + Task is executed completely.

However capability database manager can still mark a capability for deletion. Once capability is released then it will be deleted. Locking of capabilities hinders runtime deletion (aspect of QAS01) but we decided to adopt locking mechanism due to following reasons:

* + Making sure that all the capabilities mentioned in task are available all the time ensures functional correctness.
  + From the business context, capability manager rarely does deletion of a capability. Hence locking capabilities will not have much effect on runtime extensibility.

## Task Execution Architecture

* Capability deletion not possible while execution: Once the execution of a task begins, deletion of a capability is not possible. This decision helps functional correctness while execution however it hinders runtime extensibility (QAS01). When a task is executed completely, then task executor component release the lock in the capability database.
* Separate Capability Database with separate administrative interface: Once the task begins execution there is no scope for taking newly added capabilities into consideration because all the work of task composition and type checking is done by task composition component. Since capability database is independent of task execution, adding or deleting a new capability does not effect the task execution.

## Capability Addition Architecture

* Separate Capability Database with separate administrative interface: As already discussed previously, having a separate capability database with separate administrative interface helped the runtime extensibility both in task composition architecture and task execution architecture. This means adding a new capability is always possible, however for deletion there is a constraint that capability must be unlocked in the capability database.

## **QAS02 Static reusability**

## Task Composition Architecture

* Model view controller pattern for task composition interface: When a new capability is added, same type checking infrastructure can be used to modify the GUI interface to support composition of new capability.

## Task Execution Architecture

If a new capability is added in the capability database and also used in a task then following components need to be modified in task execution architecture however most of the code will be reused.

* Same capability spawner component can be reused to support new capabilities.
* Same executor can be reused to execute newly added capabilities.

## Capability Addition Architecture

* Separate capability architecture: Separate capability architecture helps in reusing the same framework to add capabilities at runtime. This capability addition framework has different components and connectors to separate the concerns. Code of these components and connectors is reused whenever a new capability is added.

## **QAS03 Static Modifiability**

## Task Composition Architecture

* Model view controller pattern: Model view controller pattern makes it easier to modify the task composition architecture. Whenever any enhancement is done in interface it can be easily done.
* Separate task accessor/writer component: A separate task accessor/writer component improves modifiability. For instance, suppose a new enhancement is to be done that requires username to be stored in the XML file. Reading and writing of user name can be easily supported in the parser of this accessor/writer component.
* Modifiability in task database: Suppose new information is to be added in the database. In such scenario, same database can be easily modified to support new information.

## Task Execution Architecture

* Event structures: The structure of event can be designed in such a way that same code for detecting and parsing the events can be reused for new events. However in case of any new data field in the event, same event parsing class can be easily enhanced to support the change.
* Separate controller is for monitoring console. Model view controller pattern helps in extending monitoring GUI. Whenever an enhancement is to be done in the monitoring GUI, changing model view and controller components can easily do it.

## Capability Addition Architecture

* Separate capability accessor/writer component: Suppose a change is done that requires developer’s information for installing capability into the system. With separate accessor/writer component it is easier to read and write developer’s information. Hence static modifiability is enhanced by separate capability accessor/writer component.
* Model view controller pattern for administrative interface: Model view controller pattern helps to easily modify the GUI administrative interface.

## **QAS04 Reliability**

## Task Composition Architecture

* Not applicable

## Task Execution Architecture

* Cobot Connection Heartbeat: Periodically messages are sent from the Cobot back to the system that ensures the connection to the Cobot is maintained. This increases reliability by adding a mechanism for monitoring the connection and reliability of the Cobot. Since connection with the Cobot is critical, the heartbeat from Cobot is send frequently then other status update events from Cobot.

## Capability Addition Architecture

* Not applicable

## **QAS04 Reliability**

## Task Composition Architecture

* Not applicable

## Task Execution Architecture

* Used call-return wherever possible:  System uses call return wherever possible to improve reliability. For instance, task executor is directly invoked by execution controller, which could also be done via event based communication.
* Reliable event based communication: Event based communication is itself unreliable by nature. However it was required to fulfill the functional requirements to report the status of capability execution to the user. But following three mechanisms are in place to make sure that event based communication is reliable:
  + Acknowledge message is send by the receiver
  + Each event is send repeatedly until acknowledgement message is received from the receiver.
  + For high priority events like user interrupt, system can have event priority-queue to avoid delay in processing them.

## Capability Addition Architecture

* Not applicable

## **QAS05 and QAS06 Performance**

## Task Composition Architecture

* Saving Meta data for XML files (QAS06): Meta data for XML files is stored, whereby increasing performance through allowing for faster loading of already composed tasks. For example, overall input and output for a task are stored as meta data, therefore we do not need to go and re-parse the whole task XML.
* Run-time type checking (QAS06): This adversely affect performance by adding another time intensive step, however this run-time type checking strongly supports QAS01 and therefore this trade-off was made.
* Use of Call-return (QAS06): This system uses call-return to communicate between its components, whereby increasing performance through reduction of steps in communication.
* Separate Task Composition Controller (QAS05): Increases performance through decoupling of controller tasks, whereby allowing for parallel processing of controllers. For example, a user could compose a new task as a previous task is still executing.

## **Task Execution Architecture (QAS05)**

* Use of Call-return: Other than the specified communications with and through the Event Communication Channel, all communication between components occurs through call-return based mechanisms. This increases performance through reduction of steps and overhead in communication between components.
* Different time durations for different types of message: For instance, runtime error message is reported more frequently by capability execution threads then the heartbeat (based on the criticality). This increases performance by reducing the overhead involved with messages.
* Use of Thread Pool: A Thread pool is used to avoid unnecessary overhead of thread creation and destruction. This provides a central location for multiple threads to be created and run in parallel.
* Use of Event Structure: Interrupts and most control mechanisms in the system work through the Event Communication Channel, whereby inhibiting performance through the use of event-based architecture.
* Separate Task Execution Controller: Increases performance through decoupling of controller tasks, whereby allowing for parallel processing of controllers.
* Use of two Priority Queues: Two priority queues are used for processing and of events, one with a higher priority and one with a lower. All interupt events are added to the higher priority queue, and all other events are stored in the lower priority queue. This decreased performance due to the need to poll, process and keep track of two queues.
* Use of Task Tree: Tasks are processed as DAG (directed acyclic graphs) and reduced into execution trees. This increases performance by generating the optimal execution of the task. This decreases performance due to the need to poll, process and keep track of two queues.
* Use of Task Tree: Tasks are processed as DAG (directed acyclic graphs) and reduced into execution trees. This increases performance by generating the optimal execution of the task.

## Capability Addition Architecture

* Not applicable.

# Appendix

1. *Task*: is a general term that describes the actual unit of work that the CoBot3 can carry out. i.e. This can be the most basic element like the capability.
2. *Capability*: is the actual java class that represents a task. Capability is built by the capability developer.
3. *Composition-Time:* This is a category that specifies the state of the system during which a user is composing a task.
4. *Execution-Time:* This is a category that specifies the state of the system during which a task is under execution.
5. *Task Composition:* This is a feature provided by the system that allows a user to chain together a set of capabilities and tasks to form a single task. For example, Find Person task can be composed by chaining together “Look up calendar”and “Go to Room”capabilities.