BalticLSC Software Design

BalticLSC Admin Tool Technical Documentation, Design of the Computation Application Language, Design of the BalticLSC Computation Tool

Version 2.0

Priority 1: Innovation

Warsaw University of Technology, Poland
RISE Research Institutes of Sweden AB, Sweden
Institute of Mathematics and Computer Science, University of Latvia, Latvia
EurA AG, Germany
Municipality of Vejle, Denmark
Lithuanian Innovation Center, Lithuania
Machine Technology Center Turku Ltd., Finland
Tartu Science Park Foundation, Estonia

BalticLSC Software Design

BalticLSC Admin Tool Technical Documentation, Design of the Computation Application Language, Design of the BalticLSC Computation Tool

Work package	WP5
Task id	A5.2, A5.3, A5.4
Document number	O5.2, O5.3, O5.4
Document type	Design Specification
Title	BalticLSC Software Design
Subtitle	BalticLSC Admin Tool Technical Documentation, Design of the Computation Application Language, Design of the BalticLSC Computation Tool
Author(s)	Agris Šostaks (IMCS), Michał Śmiałek (WUT), Kamil Rybiński (WUT)
Reviewer(s)	Edgars Celms (IMCS), Radosław Roszczyk (WUT)
Accepting	Michał Śmiałek (WUT)
Version	2.0
Status	**Final version**

EUROPEAN UNION

EUROPEAN
REGIONAL
DEVELOPMENT
FUND

History of changes

Date	Ver.	Author(s)	Change description
11.11.2019	0.1	Agris Šostaks (IMCS)	Initial structure.
12.11.2019	0.2	Agris Šostaks (IMCS)	Behavioural model added.
19.11.2019	0.3	Agris Šostaks (IMCS)	Executive summary, Introduction added
12.12.2019	0.4	Agris Šostaks (IMCS)	DTO-s, Conclusion added
17.12.2019	1.0	Agris Šostaks (IMCS)	Finalized, after review.
03.11.2020	1.1	Agris Šostaks (IMCS)	New content added.
05.11.2020	1.2	Agris Šostaks (IMCS)	Frontend Components. Diagram DTO-s. Descrip-tions.
31.03.2021	1.3	Agris Šostaks (IMCS)	CAL description added.
02.04.2021	1.4	Agris Šostaks (IMCS)	Added descriptions for component model and class model. Updated models.
12.04.2021	1.5	Agris Šostaks (IMCS)	Updated behavioural model. Minor updates.
19.06.2021	1.6	Michał Śmiałek, Kamil Rybiński (WUT)	Updated with CAL semantics
20.06.2021	1.7	Edgars Celms (IMCS)	Output review
22.06.2021	1.8	Agris Šostaks (IMCS)	Minor updates
25.06.2021	1.9	Radosław Roszczyk (WUT)	Output review
30.06.2021	2.0	Agris Šostaks (IMCS)	Final version

Executive summary

The overall aim for the Baltic LSC activities is developing and providing a platform for truly affordable and easy to access LSC Environment for end-users to significantly increase capacities to create new innovative data-intense and computation-intense products and services by a vast array of smaller actors in the region.

BalticLSC Software Design document is based on work done within activities of the BalticLSC Work Package 5 (WP5) - Design of the BalticLSC Admin Tool (A5.2), Design of the Computation Application Language (A5.3), and Design of the BalticLSC Computation Tool (A5.4). It involves also workshops and technical sessions with participation of external experts and led mainly by the project's technology partners from - Warsaw University of Technology (WUT), Institute of Mathematics and Computer Science, University of Latvia (IMCS), RISE Research Institutes of Sweden AB (RISE). All workshops and meetings (in-person and remote) are held during Q1-Q2-Q3-Q4 2019 and Q1-Q2-Q3-Q4 2020.

BalticLSC Software Design document contains technical design for BalticLSC Software. It is intended to use by BalticLSC technology partners responsible for the development of BalticLSC Software.

Table of contents

1. Introduction

1.1 Objectives and scope

The objective of the document is to describe the design of the BalticLSC Software. The goal is to have the BalticLSC Software design specification which is usable for the BalticLSC Software Development. The technical documentation for the Computation Tool, Admin Tool, and CAL will consist of several elements. It provides detailed structural models of the designed software tool - several component and class models have been developed, reflecting the high-level componentization of software and low-level solutions regarding each of the component internals. These models use well-established software design principles, pertaining to software layering, communication between components and software development frameworks.

Based on the component models, the behavioural models describe dynamics of interactions between and within the computational components. This part of design base on the best practices and industry solutions regarding patterns of interactions between components.

The document provides comprehensible and repeatable solutions to the requirements regarding definition and execution of LSC applications. For this purpose, it uses industry-standard - UML.

1.2 Relations to Other Documents

The current BalticLSC Software Design document provides the design specification of the BalticLSC Software for requirements specified in the O3.3: BalticLSC Software User Requirements Specification. It is the joint document for three outputs – O5.2B: BalticLSC Admin Tool Technical Documentation, O5.3B: Computation Application Language Manual, and O5.4B: BalticLSC Computation Tool Technical Documentation. We decided to have a single document which has complete design specification for all three initially planned software parts – Admin Tool, CAL Language and Computation Tool. Since all three parts are heavily interconnected, the design of software has been done (and still is being done) simultaneously. Thus, we do not explicitly divide the design document into such three parts, but rather we do split it according to the types of models used – firstly, we define the main components using component model (UML Component and Class diagrams, see Chapters 2), secondly, we define the conceptual classes of the BalticLSC Software using class model (UML Class diagrams, see Chapter 3), and thirdly, we define the behaviour of the main components using behavioural model (UML Sequence, UML Activity, and UML State diagrams, see Chapter 4). Rigorous description of abstract and concrete syntax of Computation Application Language is done using metamodeling approach (See UML Class diagram in Chapter 5) All models are constantly updated, and the "live" version of the design model is maintained by Warsaw Technical University (WUT) as an UML model. It is publicly available in the project's website https://www.balticlsc.eu/model/

1.3 Intended Audience and Usage Guidelines

This Design specification is the Outputs 5.2B, 5.3B and 5.4B as described in the Baltic LSC Application Form and intended for internal use within BalticLSC consortium as basis for future software development activities as well as for reporting purposes for local First Level Control and Baltic Sea Region Program. It is also meant for use by various stakeholders in the Baltic region, e.g., software innovator SMEs, research groups, individual software developers and all interested in technical details of tools for developing and executing LSC applications within large scale computation environments, to build compatible solutions and integrate their systems into the BalticLSC Network.

2. Component Model

This chapter contains the detailed structural model of the designed software tools, namely, the component diagrams describe the main components and class diagrams describe the structure, API interfaces and data-transfer objects (DTO) of these components. Structural models reflect the high-level componentization of software and low-level solutions regarding each of the component internals. Industry standard - UML Component and UML Class diagrams have been used to provide comprehensible and repeatable solutions.

Figure 1, Overview of the BalticLSC Software shows the main components of the BalticLSC Software. They are:

Master Node is a software component which provides BalticLSC Software user interface via **FrontEnd**: CAL Editor, App Store, Computation Cockpit, Data Shelf, Resource Shelf, Development Shelf, and implements the main functionality of the BalticLSC Software via **Master Node Backend**: computation application compilation and distribution to computation resources, computation application management, user management and security, etc. Master node is securely accessed using web browser from the client device. Communication between Frontend and Master Node Backend components is done using REST protocols. Master Node Backend distributes the particular computations to the available computation resources.

Cluster Node is software component which is installed on the particular computation resource within Kubernetes cluster. It allows execution of the particular computations, communication between computation modules and Master Node. The main components of the Cluster Node are: Batch Manager, Cluster Proxy and Module Proxies. Actual computations are run within a particular Kubernetes namespace as docker containers and we call them Job Instances.

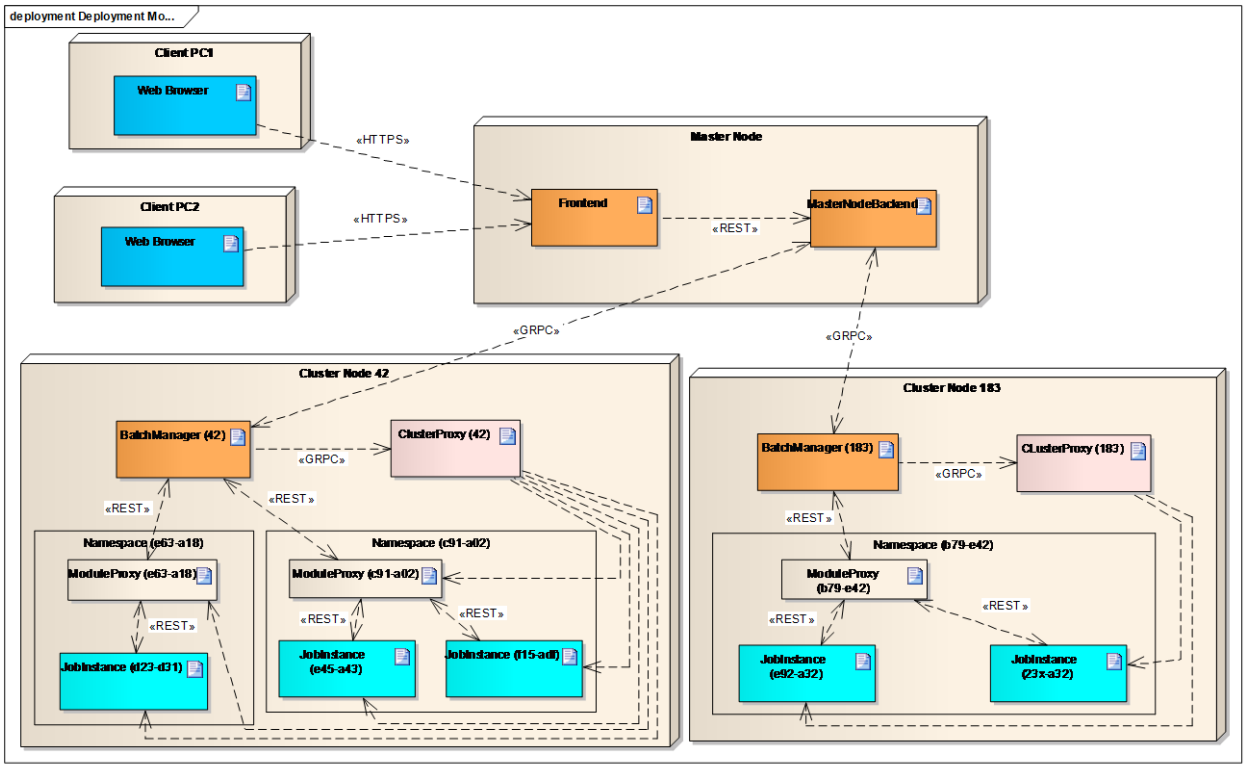

Figure 1, Overview of the BalticLSC Software

Next, the detailed models of both components of BalticLSC Software, namely, Master Node and Cluster Node are described. The internal structure is described for each component using UML Component diagrams. Next, each sub-component is described using set of UML class diagrams depicting classes,

interfaces, their dependencies, generalizations and relations with other components. Several stereotypes have been used:

GRPC controller – a class which is implemented as a provider of remote procedure call system *gPRC*.

REST controller – a class which is implemented as a provider of web services using representational state transfer (REST) system.

internal interface – an interface which describes the functionality provided by a software component.

GRPC proxy – a class which is implemented as a proxy service for GPRC controllers.

REST proxy – a class which is implemented as a proxy service for REST controllers.

API – a class which implements some functionality.

data transfer – a Data Transfer Object (DTO) class which describes structure of object used as request or response parameter by REST and GPRC controllers.

API controller – a class which is implemented as a provider of functionality.

data access – a Data Access Object (DAO) class which is used to access data stored in the relational database.

trace – a dependency which traces a class in the component model (e.g., data transfer class) to the class in the conceptual class model.

infrastructure – a class which provides access to the software infrastructure, e.g., relational database.

2.1 Cluster Node (Cluster)

Cluster Node software is installed in the Kubernetes (or Docker Swarm) cluster. The purpose of this software is to allow Mater Node to manage the actual computations and provide resource usage statistics for these computations using **Baltic Node API** (see Figure 2, Components of Cluster Node). There can be multiple Cluster Nodes in the BalticLSC Network. Every cluster has a single Batch Manager installed. Cluster is a BalticLSC Platform software installed in the Kubernetes cluster which provides all low-level functionality to the Batch Manager via **Cluster Proxy API**. Cluster manages Docker Containers – Job instances that provide **Job API** and use **Tokens API** implemented by Batch Manager to communicate the state of the computations.

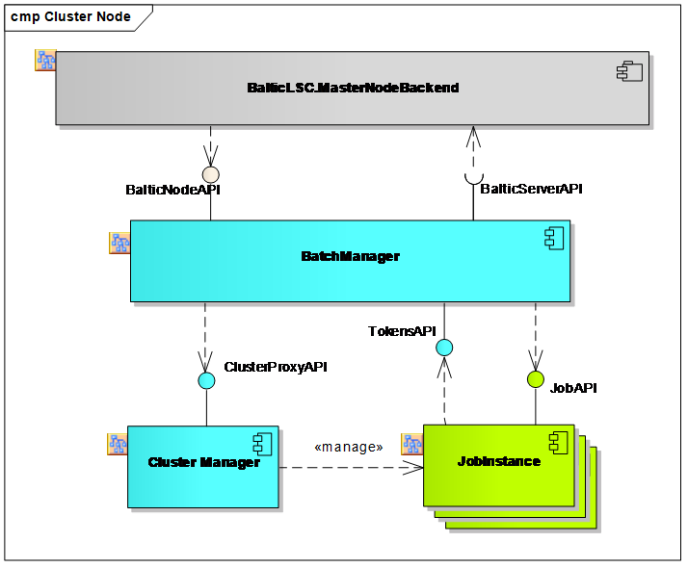

Figure 2, Components of Cluster Node

2.1.1 [O5.4] Batch Manager

Batch Manager (see Figure 3) provides functionality to get statuses of jobs, finish jobs and job batches (`IBalticNode`), consume batch and execution messages (tokens) to ensure execution of the task (`IMessageConsumer, ITokens`). It is responsible for managing the node's activities: managing jobs assigned to the current node and managing availability of the whole node. It is also responsible for constant monitoring of the statuses of jobs, and signaling their status to the Master Node.

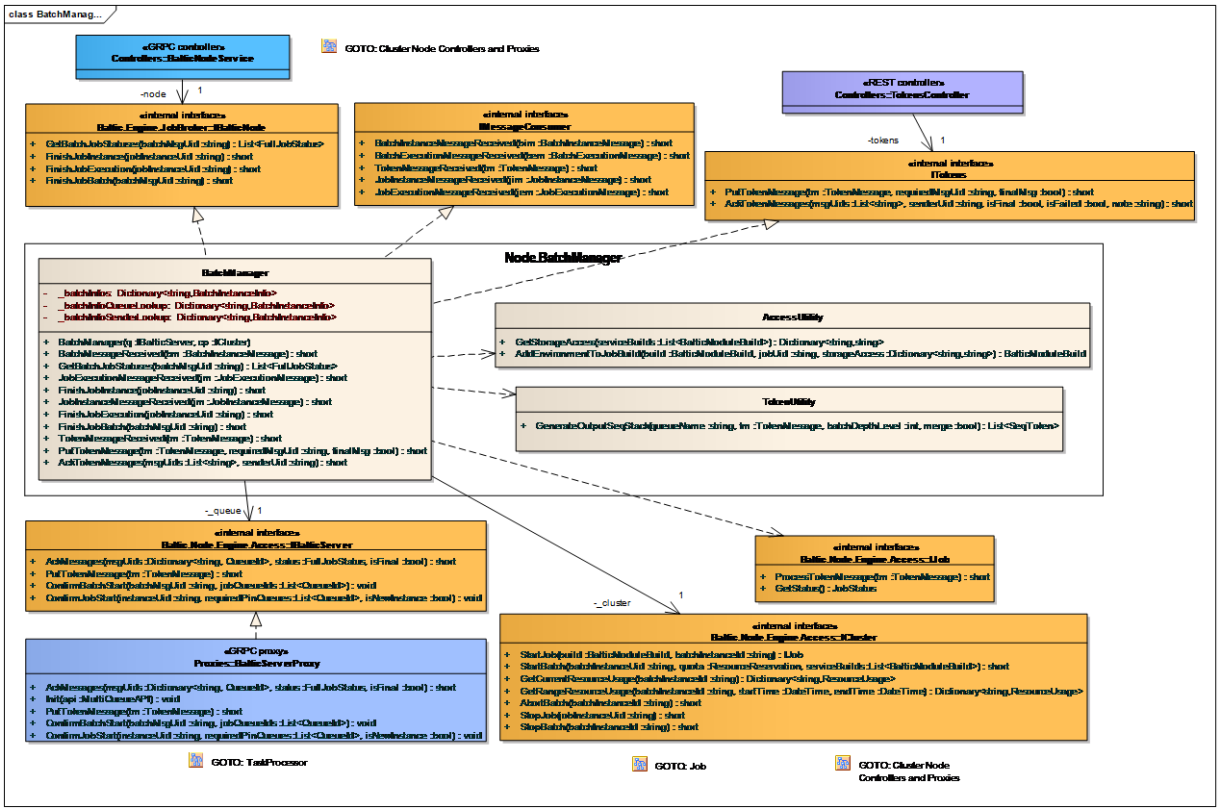

Figure 3, Batch Manager

2.1.2 [O5.4] [O5.2] Cluster Node Controllers and Proxies

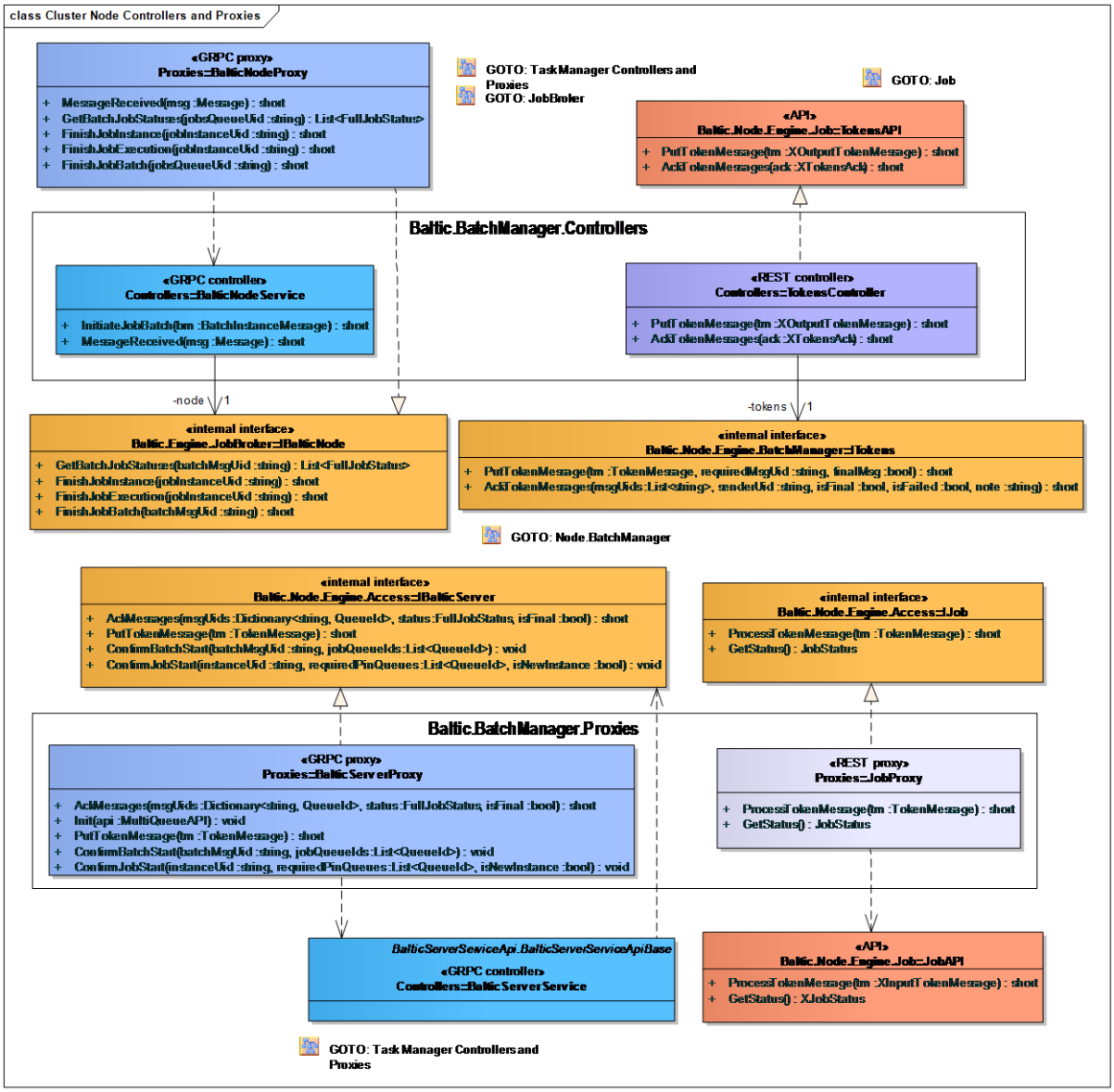

Figure 4, Cluster Node Controllers and Proxies

2.1.3 [O5.4] Cluster Node Data Transfer Objects

Figure 5 describes the DTO-s used to communicate between Job Instances and Batch Manager. Basically, they are messages containing lists of tokens to be executed or already executed, as well as acknowledgment messages.

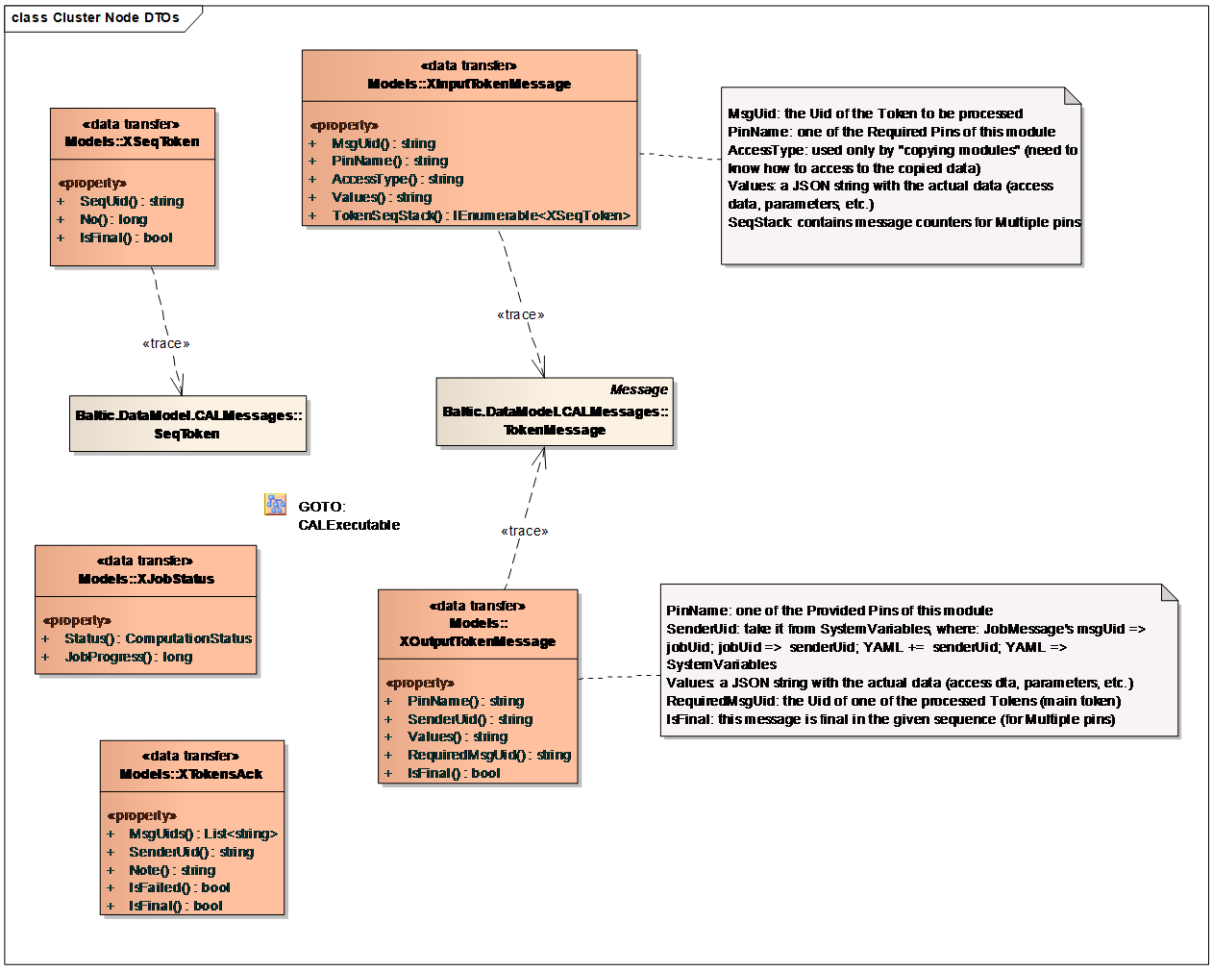

Figure 5, Cluster Node DTO-s

2.1.4 [O5.4] Cluster

Cluster is a BalticLSC Platform software component accessed by BalticLSC Software using **Cluster Proxy API**. It provides low-level operations with Kubernetes cluster, like, actual running of Docker container, creating and purging workspace (namespace), getting resource usage statistics. Cluster is responsible for management of containers within the Kubernetes cluster and monitoring of container statuses.

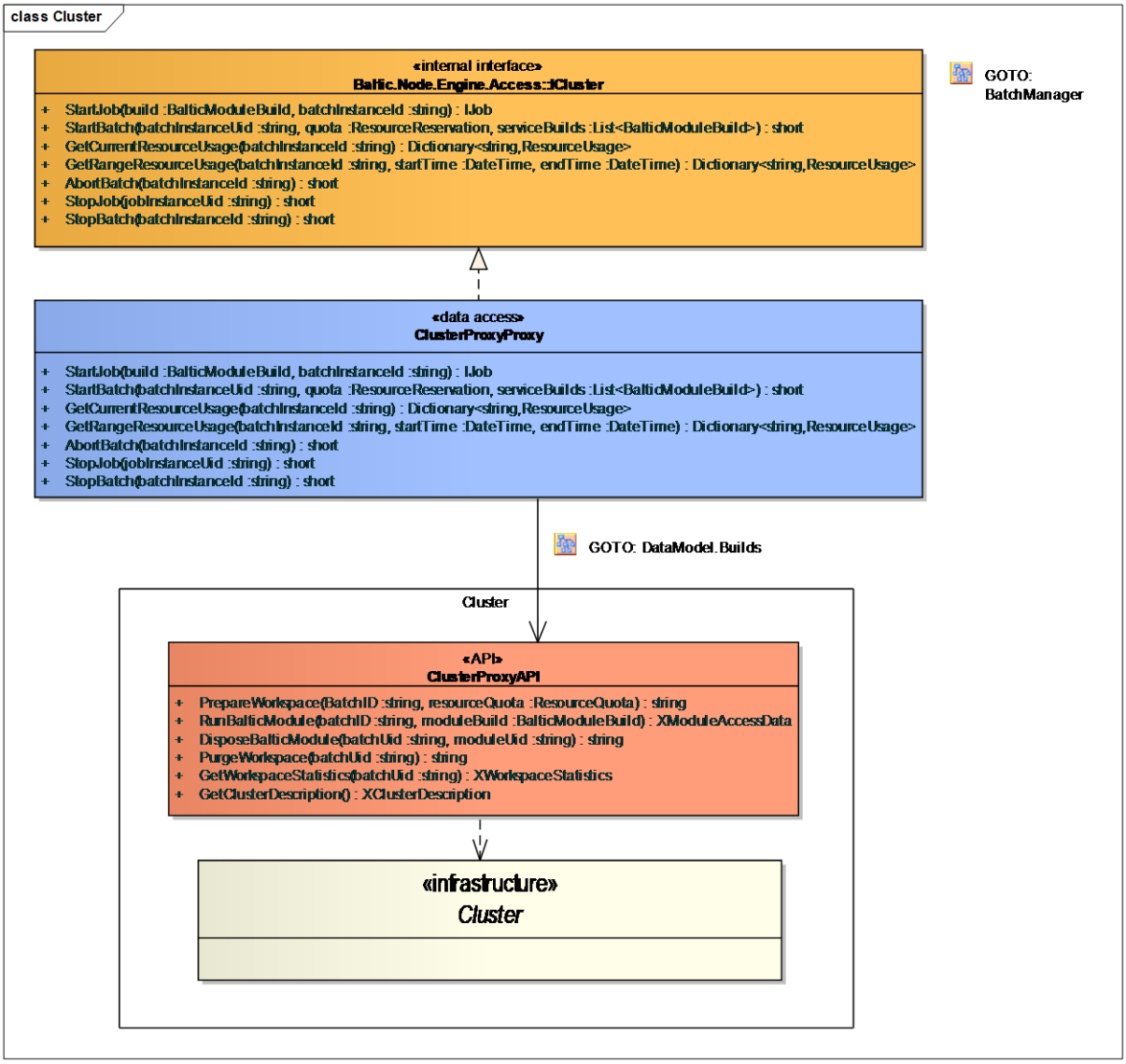

Figure 6, Cluster Proxy API

2.1.5 [O5.4] Job Instance

Job Instance is a Docker container installed in the cluster by the Cluster Proxy and performing the actual computations, according to the appropriate Computation Module's code. Batch instance is a set of Job Instances that work in the same cluster and the same workspace. Each Job Instance should implement the **Job API (Job Controller)** (See Figure 8). It provides status of the job and allows to consume the app execution token. On the other hand, a Job Instance should use the **Tokens API** to acknowledge the Batch Manager about status changes.

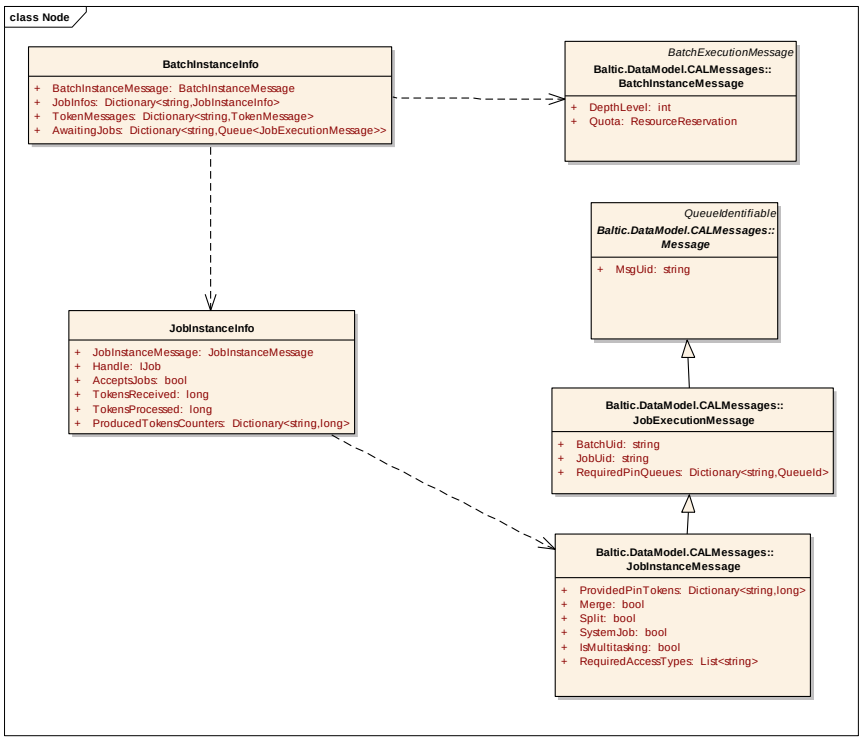

Figure 7, Job Instance

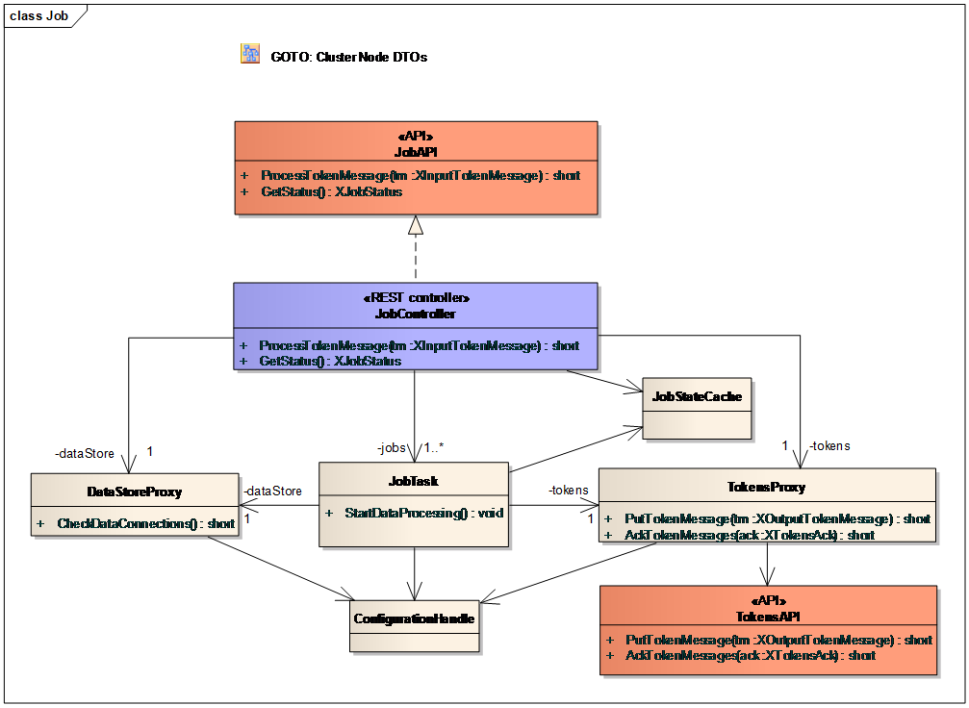

Figure 8, Job Controller

2.2 Master Node

Master Node is a central access point of the BalticLSC Network – its face and brain. It provides user interface (via **Frontend**) and functionality (via **Backend**).

2.2.1 Master Node: Backend

The main components of the Backend are divided into two categories (see Figure 9) – grey boxes are components which implement the BalticLSC Network functionality, but blue boxes are components for the infrastructure – data registries and utilities.

The main components of the Backend are:

- [O5.4] **Task Manager** allows to manage tasks and datasets.
- [O5.2] **Unit Manager** allows to create and manage CAL applications and modules.
- [O5.2] **Network Manager** allows to manage network resources.
- [O5.4] **Task Processor** allows to run computation tasks.
- [O5.4] **Job Broker** allows to distribute computations to network resources.

The main infrastructure components are:

- [O5.4] **Diagram Registry** stores CAL diagrams.
- [O5.2] **Unit Registry** stores CAL apps and information on modules.
- [O5.2] **Task Registry** stores task and dataset information.
- [O5.2] **Network Registry** stores computation resource and cluster data.
- [O5.4] **MultiQueue** provides message queue for computation execution.

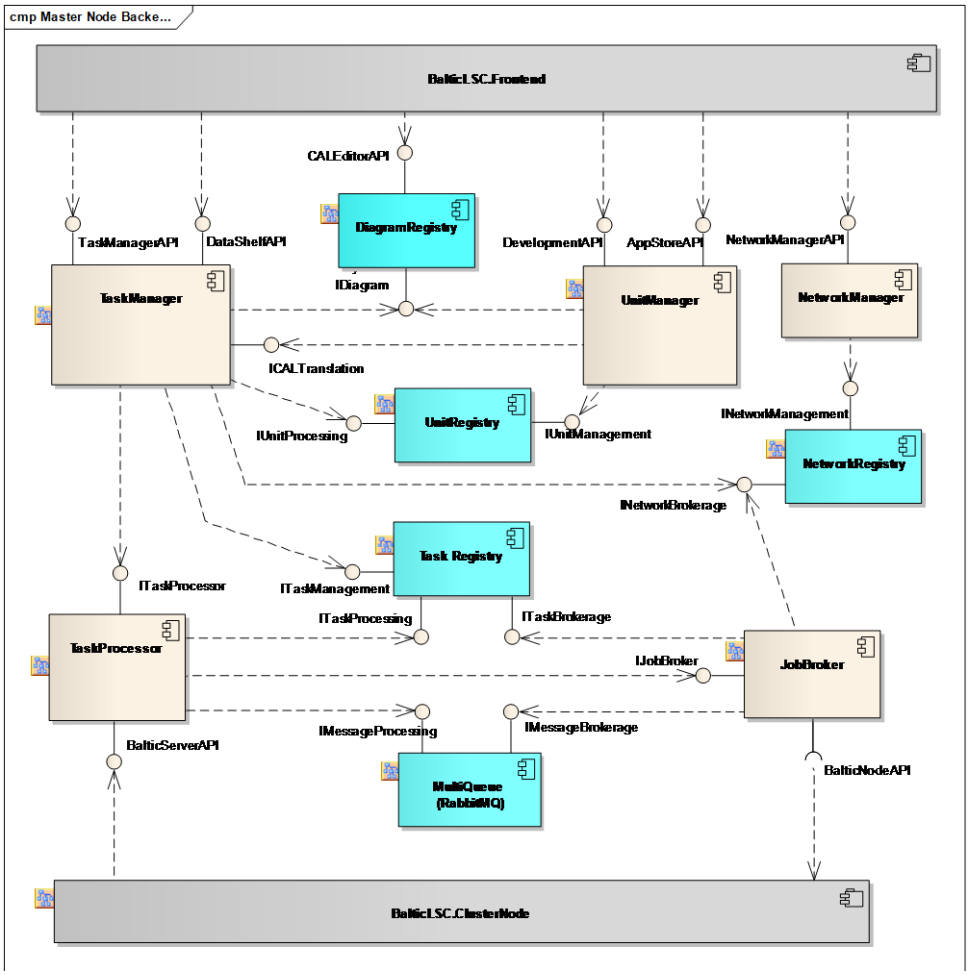

Figure 9, Components of Backend

2.2.1.1 [O5.4] Diagram Registry

Diagram Registry is a component which stores the CAL diagrams. Its purpose is to provide basic operations with diagrams – copy, create and access via `IDiagram` interface. Diagrams are the source for translation to the CAL programs (in abstract syntax). Frontend, in particularly, the CAL Editor uses `CALEditorController` to store updates to diagrams. `XDiagram` DTO is depicted in Figure 11, Diagram DTO-s.

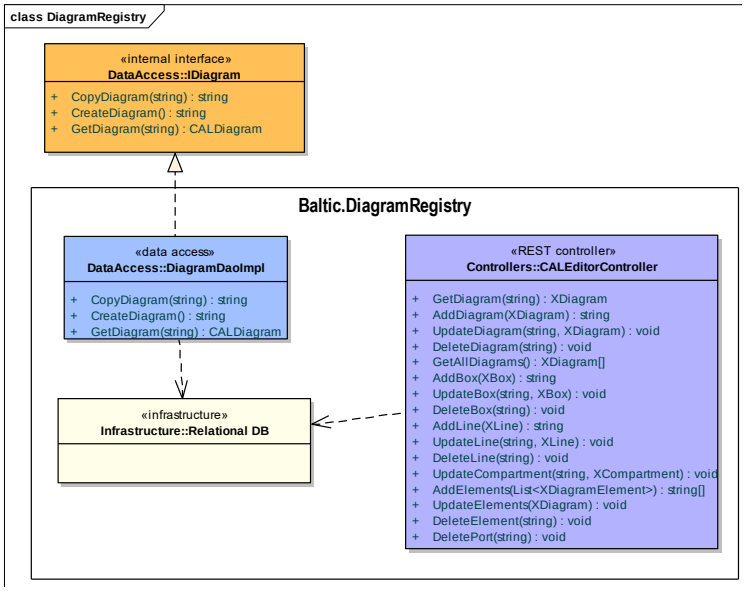

Figure 10, Diagram Registry

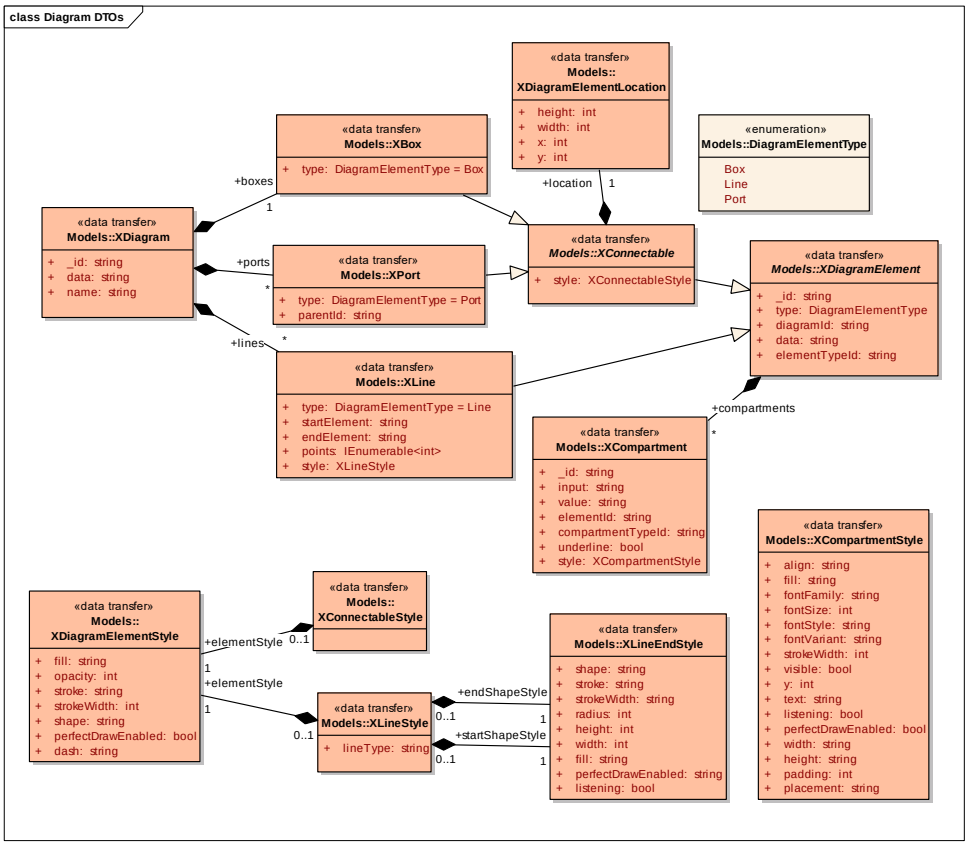

Figure 11, Diagram DTO-s

2.2.1.2 [O5.4] Job Broker

Job broker is a component which is responsible for actual brokerage of jobs and batches. It starts, finishes, update statuses of jobs and batches. Job broker distributes computation tasks (jobs) to proper computation resources.

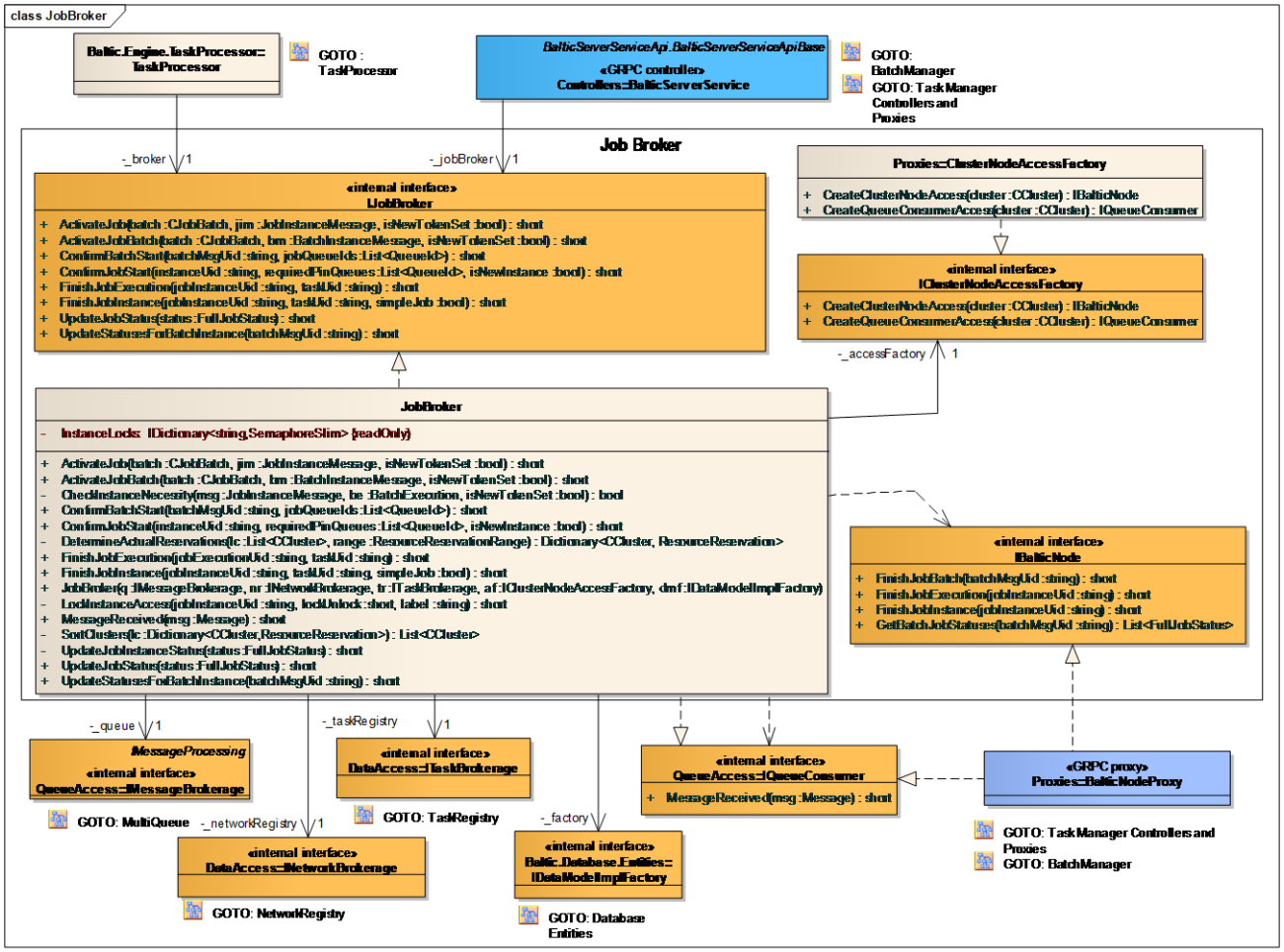

Figure 12, Job Broker

2.2.1.3 [O5.4] Multi Queue

Multi queue is a component which is responsible for communication between batch manager on the cluster node and software components responsible for processing tasks and brokering jobs, namely, the Task Processor and Job Broker. Communication is done using message queues tied to the task.

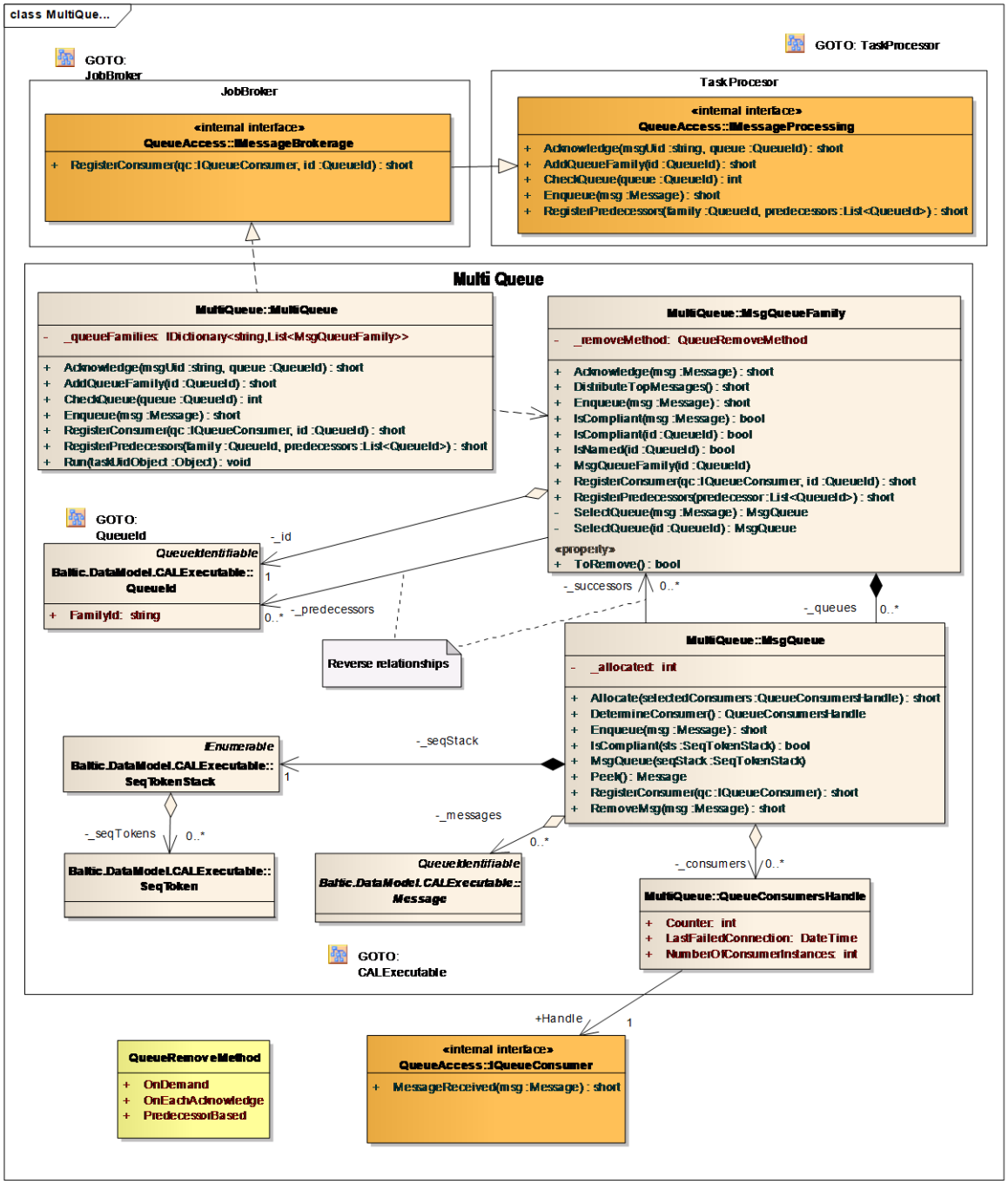

Figure 13, Multi Queue

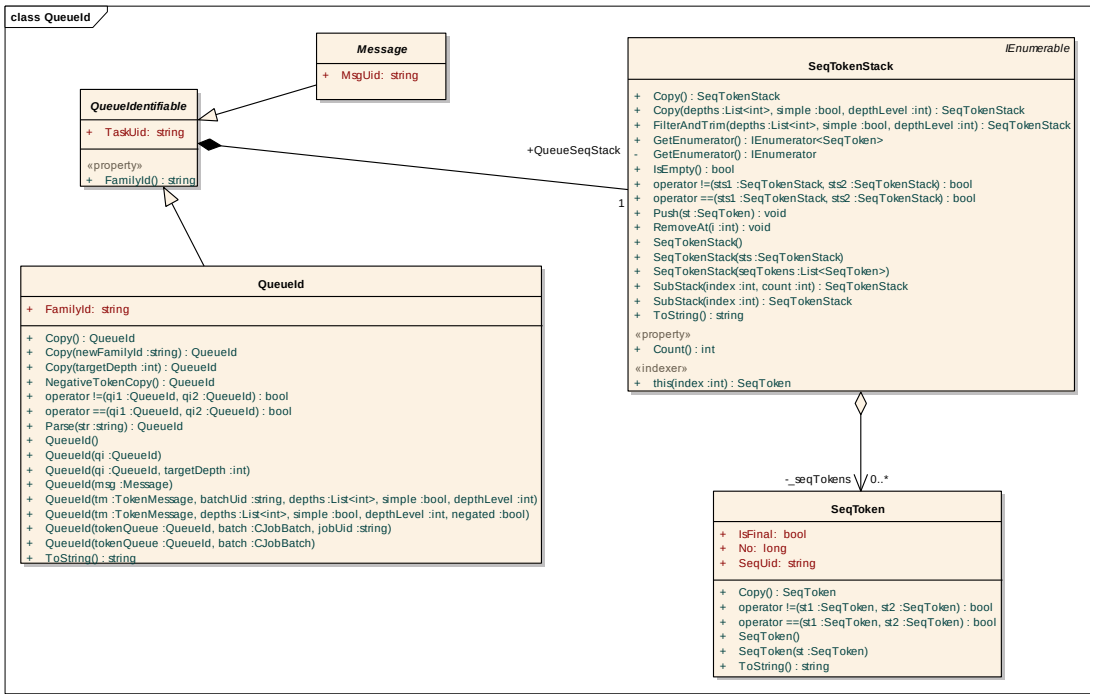

Figure 14, Queue Identifiable

2.2.1.4 [O5.2] Network Manager

Network manager is responsible for managing all the computation clusters and nodes within the network. It allows the user to register, edit and manage the available resources.

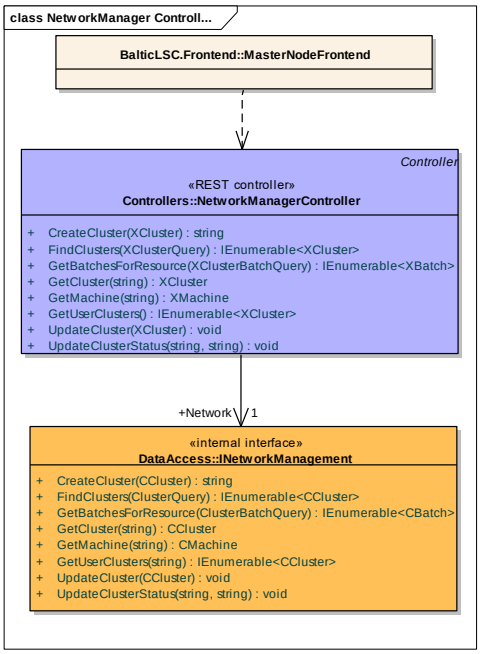

Figure 15, Network Manager Controllers

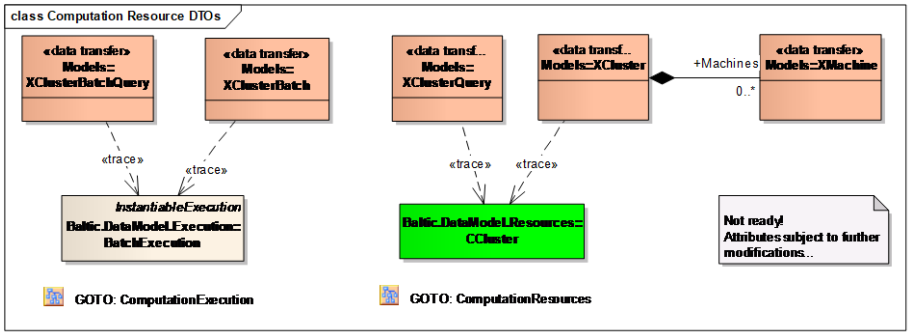

Figure 16, Network Manager DTO-s

2.2.1.5 [O5.2] Network Registry

Network registry stores the information on clusters and machines. It provides options to search for clusters with matching resources.

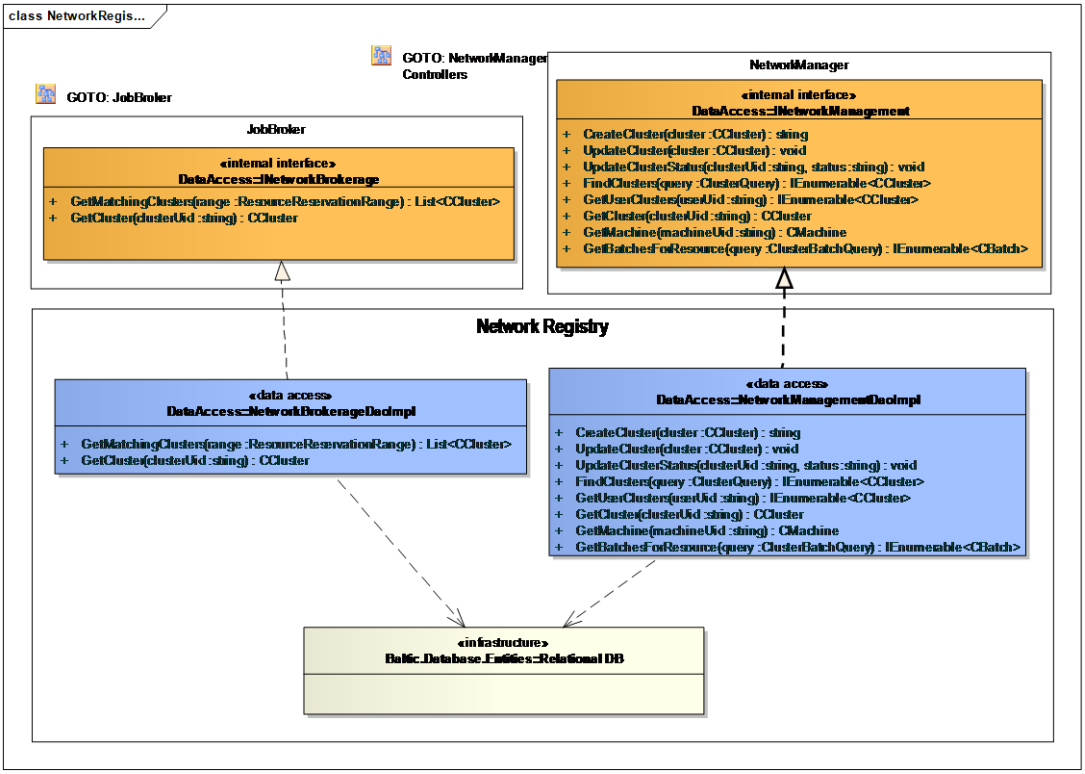

Figure 17, Network Registry

2.2.1.6 [O5.4] Task Manager

Task manager is responsible for managing tasks, as requested by the users. It also includes translation of CAL diagrams to the abstract syntax and further to `CAL Executable` language (it is done upon the first request to run the app release). Task manager is also responsible for management of the users dataset information (but not the datasets themselves).

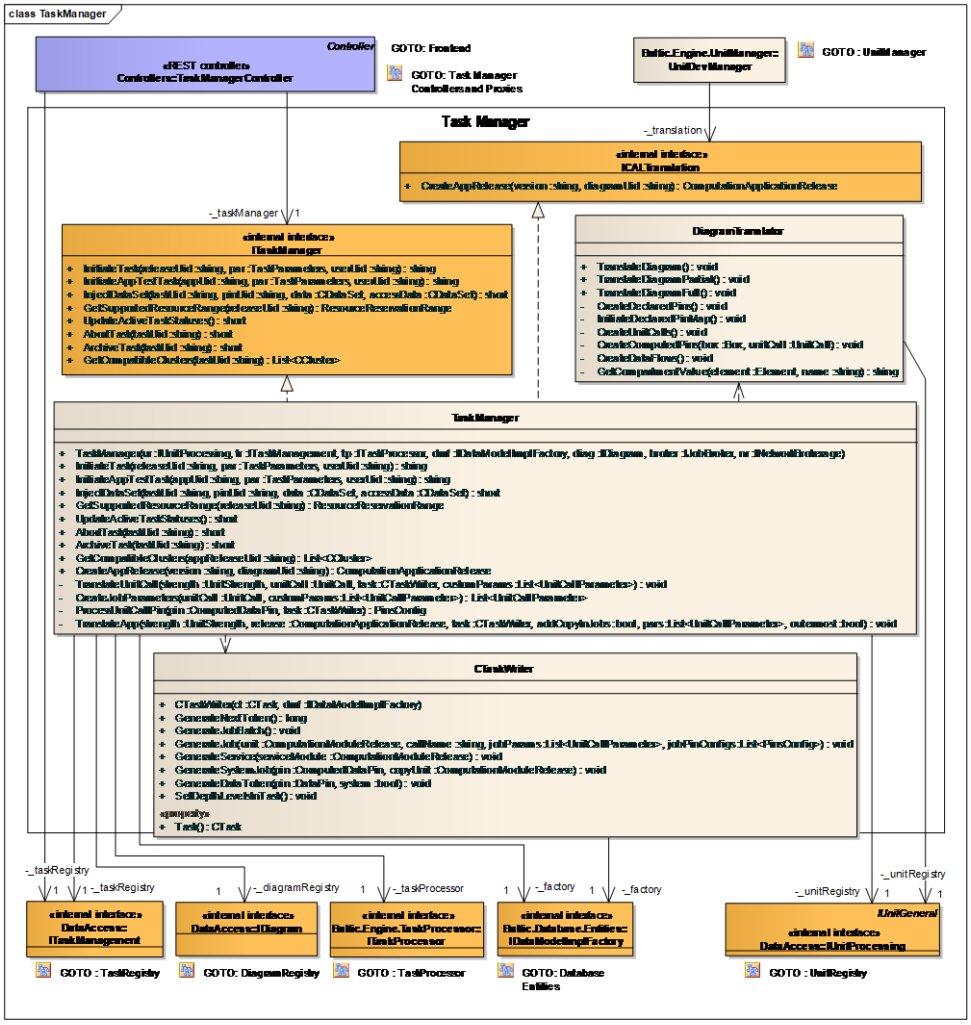

Figure 18, Task Manager

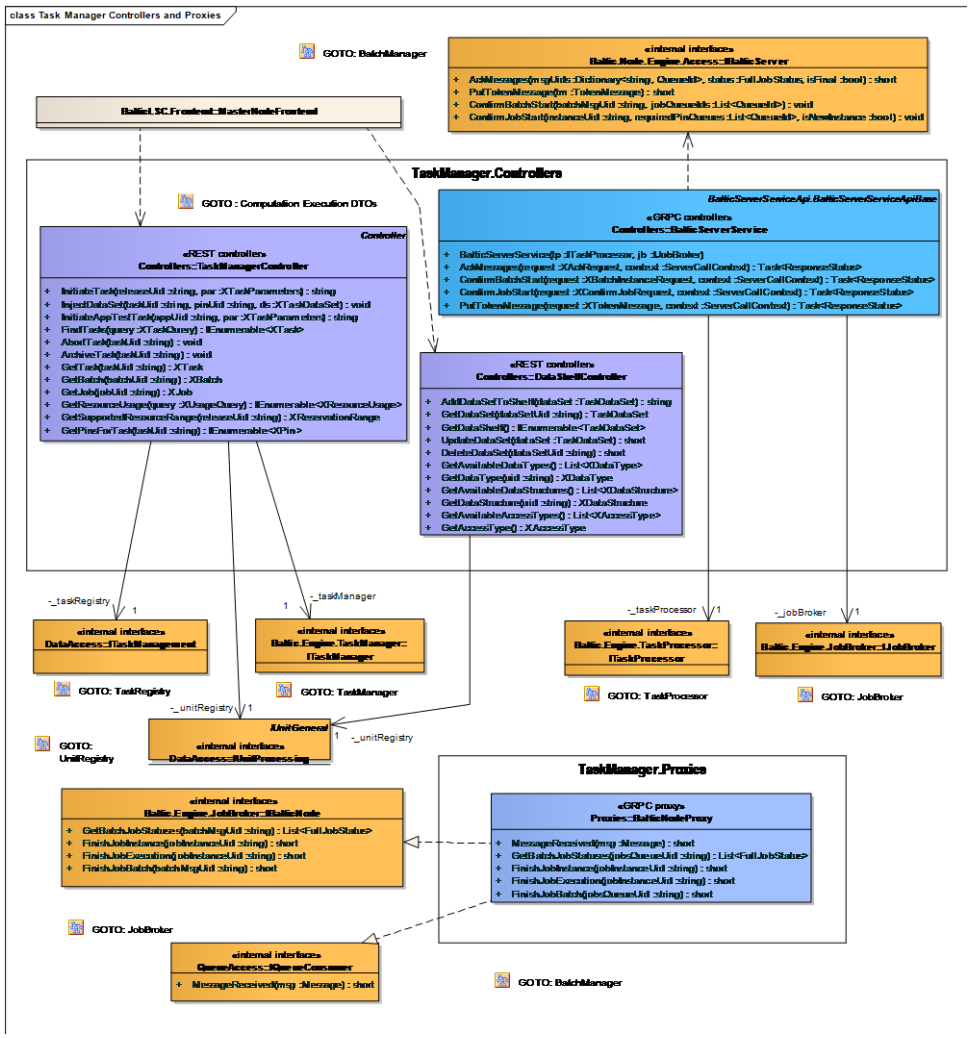

Figure 19, Task Manager Controller and Proxies

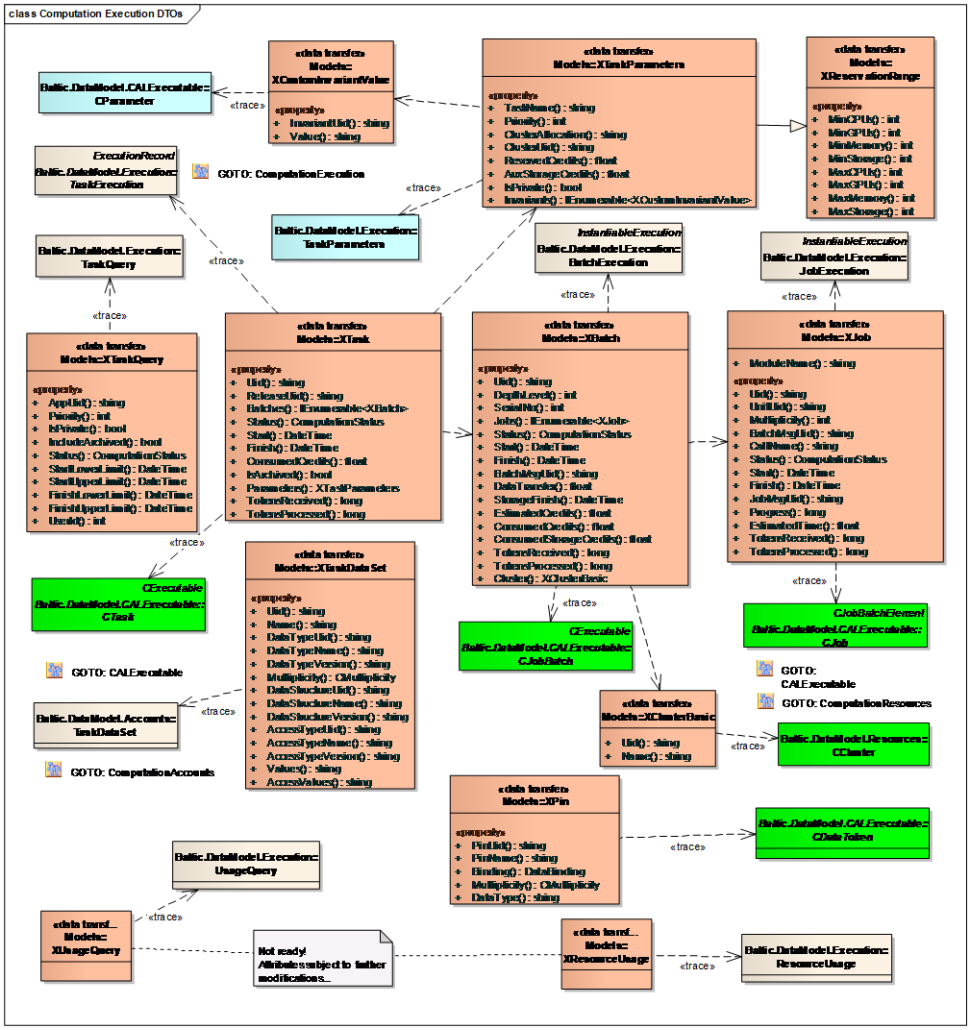

Figure 20, Computation Execution DTO-s

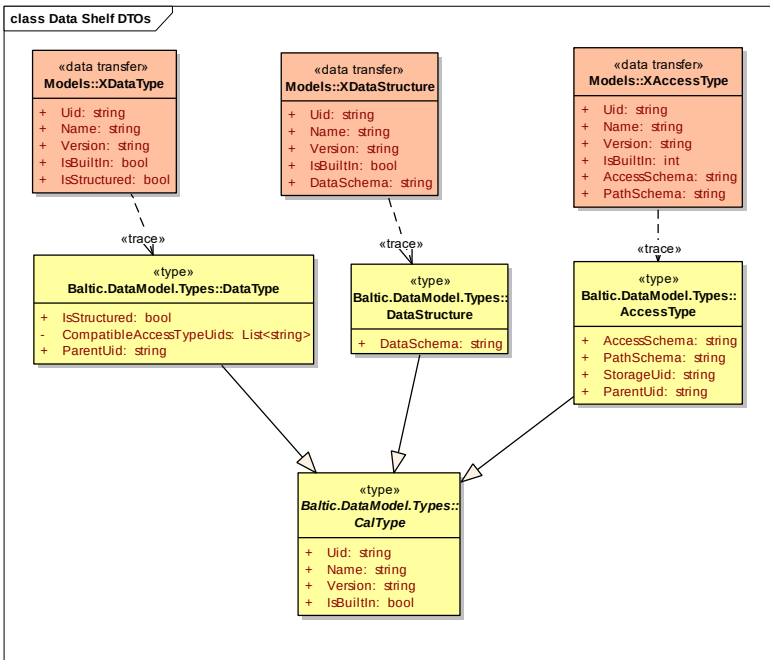

Figure 21, Data Shelf DTO-s

2.2.1.7 [O5.4] Task Processor

Task processor is responsible for execution of computation tasks. It interprets CAL Executable programs and receives messages from batch managers within clusters. Task processor determines the need for job to be done and sends all the jobs for actual execution to the Job Broker.

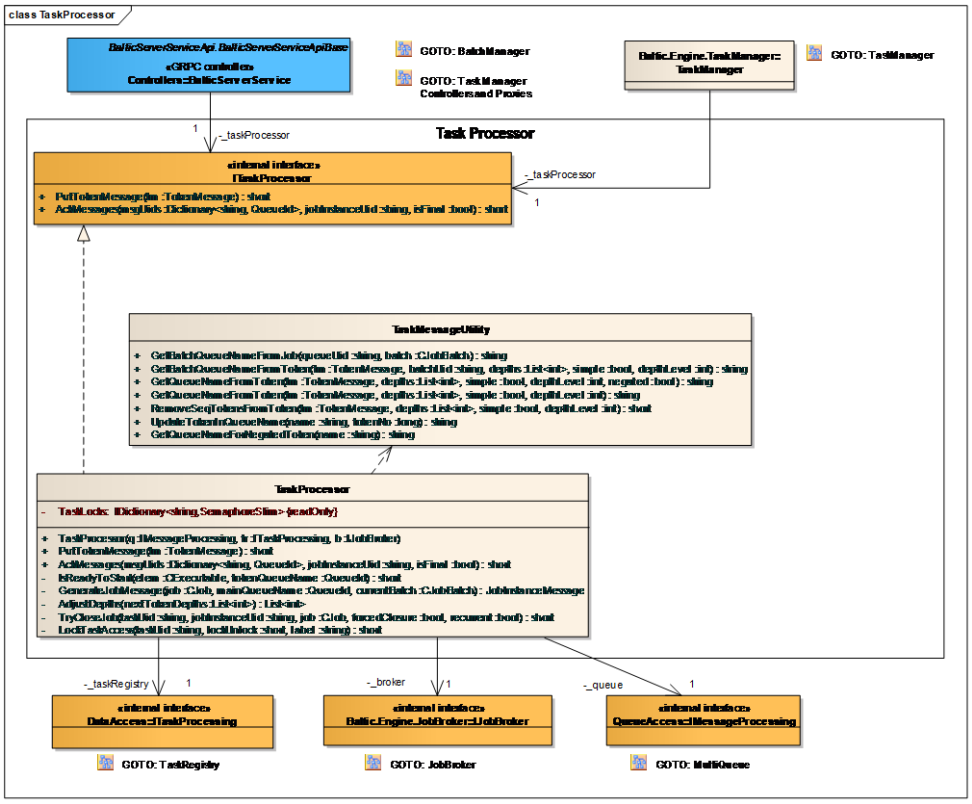

Figure 22, Task Processor

2.2.1.8 [O5.2] Task Registry

Task registry stores the information on computation tasks. It includes information on execution of tasks – details on batches, jobs and used resources.

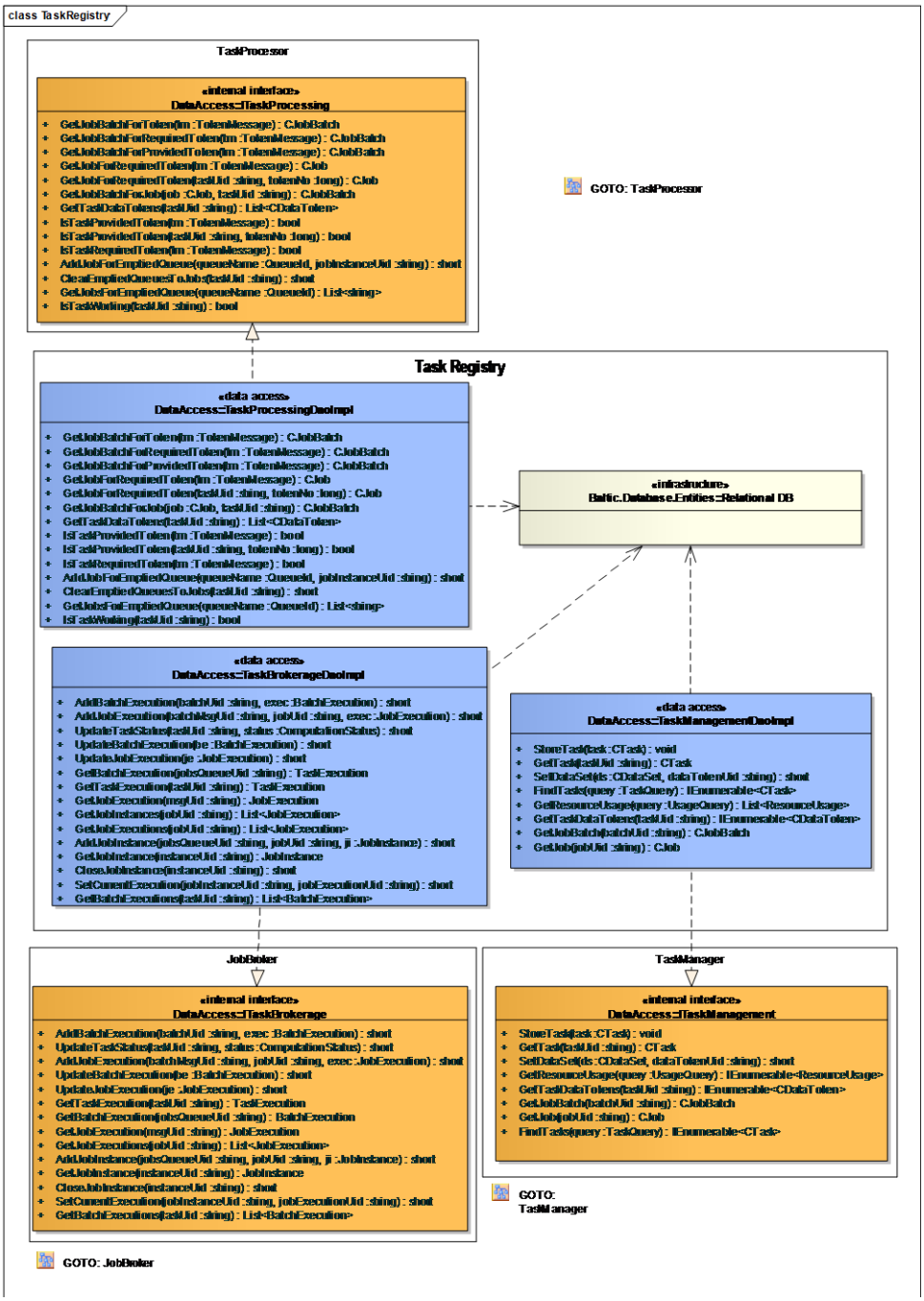

Figure 23, Task Registry

2.2.1.9 [O5.2] Unit Manager

Unit manager is responsible for managing computation units, both applications and modules. It allows creating and editing meta-information on computation units (but not "code" of the units). Unit manager is responsible for managing App-Store and user's App-Shelf.

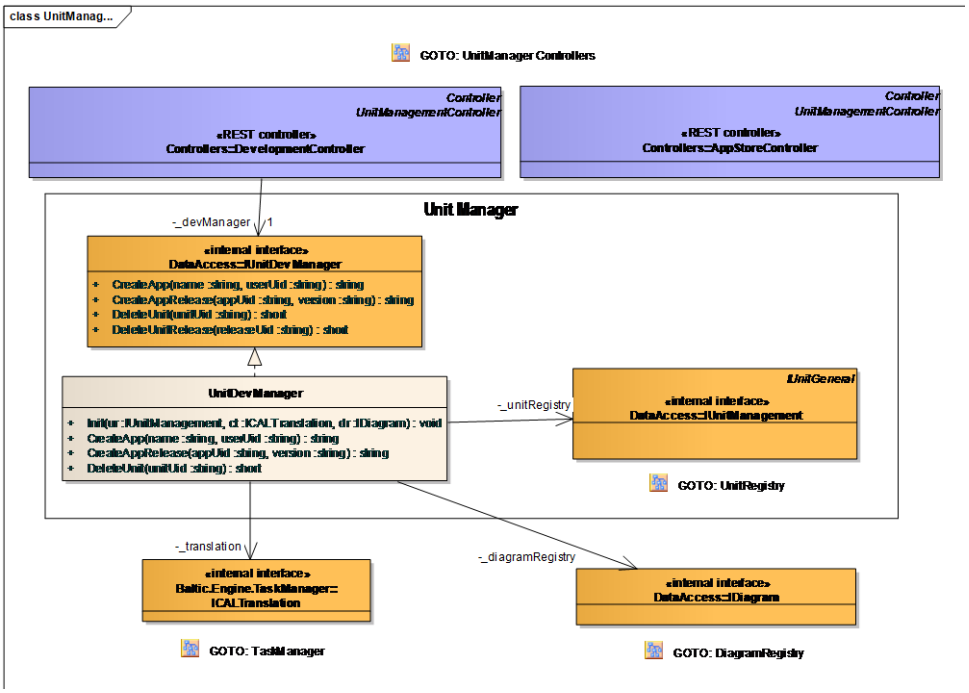

Figure 24, Unit Manager

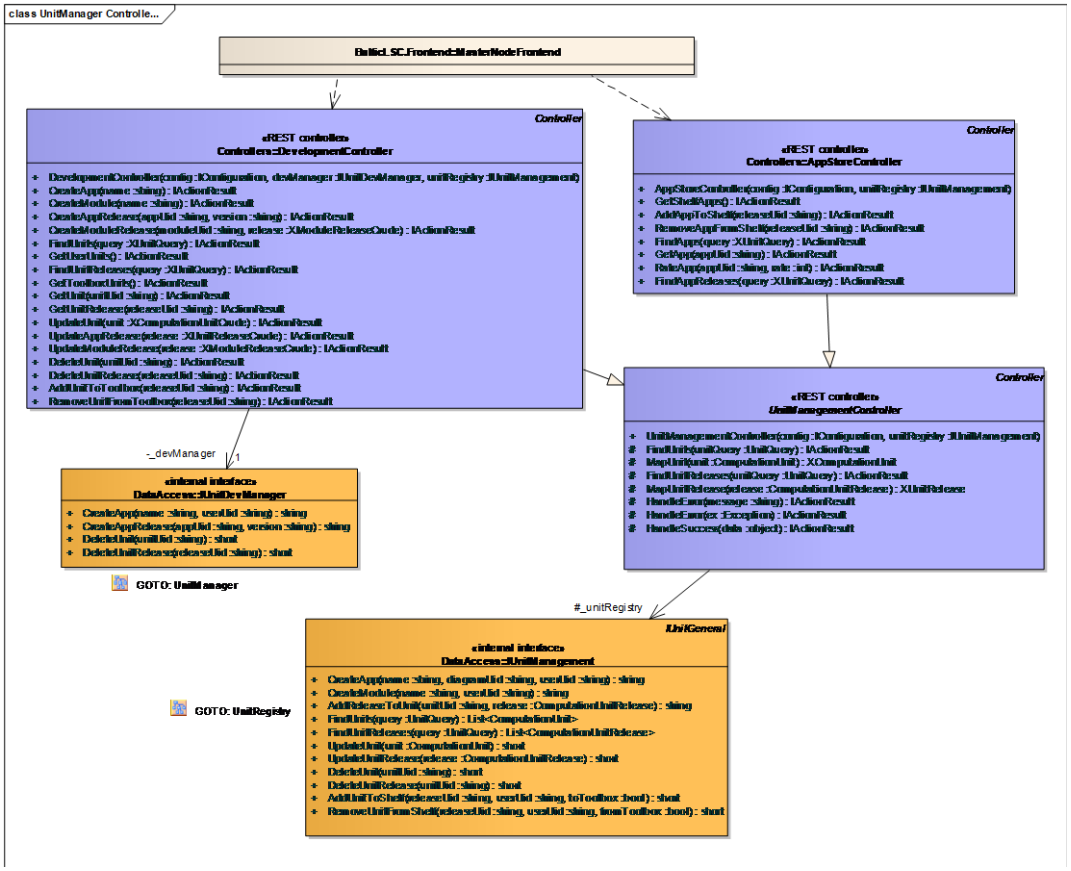

Figure 25, Unit Manager Controllers

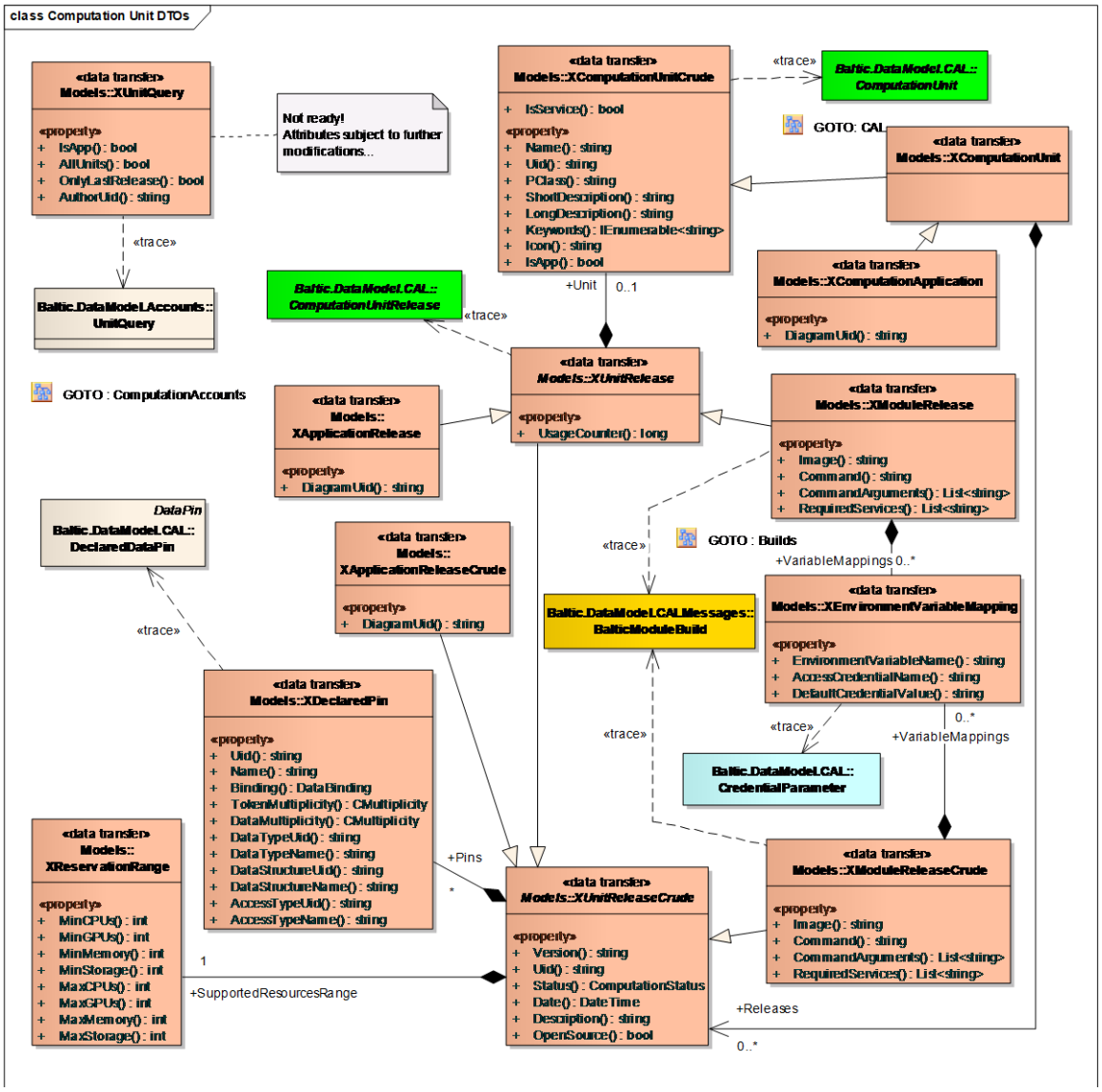

Figure 26, Unit Manager DTO-s

2.2.1.10 [O5.2] Unit Registry

Unit registry stores information on computation units.

Figure 27, Unit Registry

2.2.2 Master Node: Frontend

Frontend provides user interface to the various actors of the BalticLSC Network. Frontend is split in several components:

- [O5.2] **Administrator Cockpit** is a component for user interactions for BalticLSC network administration, e.g., user management, computation resource supervision, etc.
- [O5.2] **Resource Shelf** is a component for user interactions for management of the computation resources and clusters.
- [O5.2] [O5.4] **Development Shelf** is a component for user interactions for management of the developed Computation Applications and Computation Modules.
- [O5.4] **CAL Editor** is a component for editing, validating, and debugging CAL Applications in a graphical way.
- [O5.2] **App Store** is a component for user interactions for obtaining computation application releases and information on them.
- [O5.2] **Data Shelf** is a component for user interactions for dataset management.
- [O5.4] **Computation Cockpit** is a component for user interactions for managing (running) computation tasks.

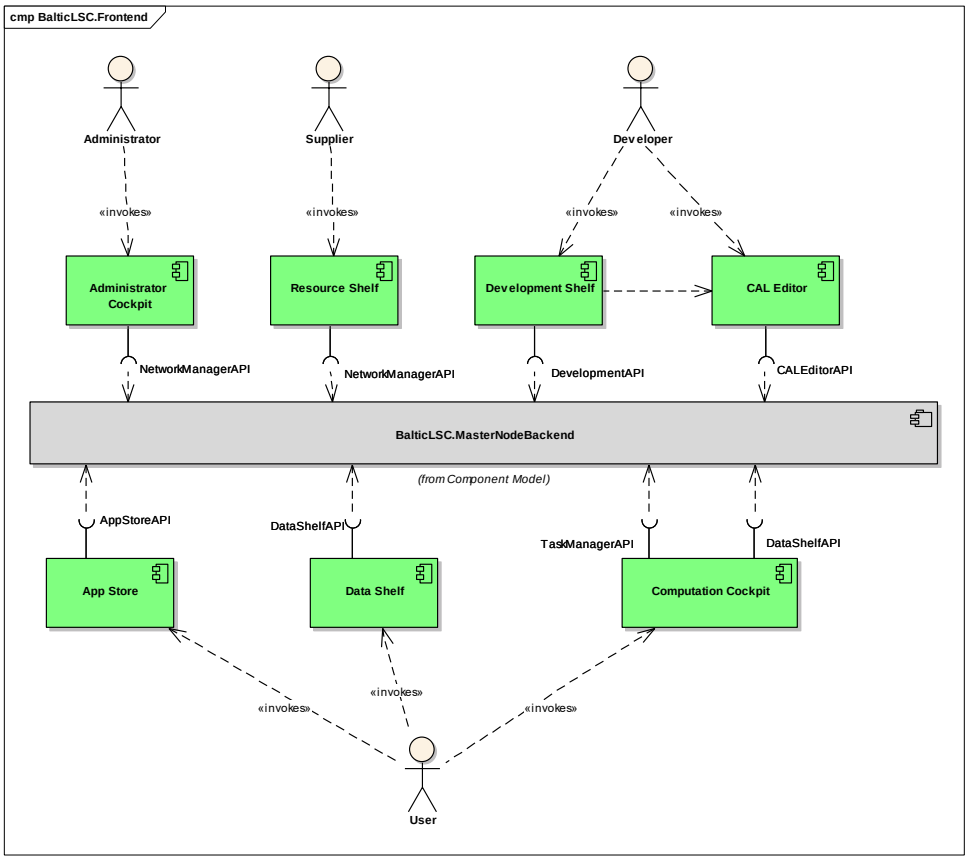

Figure 9. Component Diagram for the FrontEnd

3. Class Model

Class model describes the main concepts of the BalticLSC Software. Concepts are depicted using UML Class diagrams. These diagrams are used to generate classes for the actual software components. The classes are divided in logical packages each describing some aspects of the BalticLSC Software. These aspects are:

- [O5.2] User accounts – classes describe users and artifacts which belong or are related to the user.
- [O5.3] CAL – classes describe Computation Application Language.
- [O5.4] CAL Executable – classes describe CAL Executable language.
- [O5.4] CAL Messages – classes describe communication messages (token messages, job and batch execution messages, job, and batch instance messages) and queues which are used to handle them.
- [O5.4] Diagram – classes describe the diagram structure used to depict the CAL programs.
- [O5.4] Execution – classes describe task execution records. This includes task, batch, and service execution instances.
- [O5.2] Resources – classes describe clusters and resources within BalticLSC Network.

3.1 [O5.2] User Accounts

`UserAccount` class represents user accounts of BalticLSC Software. Each user account has:

- `ComputationUnitRelease` instances linked by `Toolbox` association – toolbox - computation unit releases used in the user's CAL Programs as Unit Calls.
- `ComputationUnitRelease` instances linked by `AppShelf` association – computation unit releases which can be used to start a new computation task.
- `ComputationUnit` instances which have `AuthorUid` set to the user's `Uid` – computations units developed by the user.
- `TaskDataSet` instances linked by `DataShelf` association – user's owned datasets which can be passed to computation tasks.
- `CTask` instances linked by `Computations` association – user's-initiated tasks.

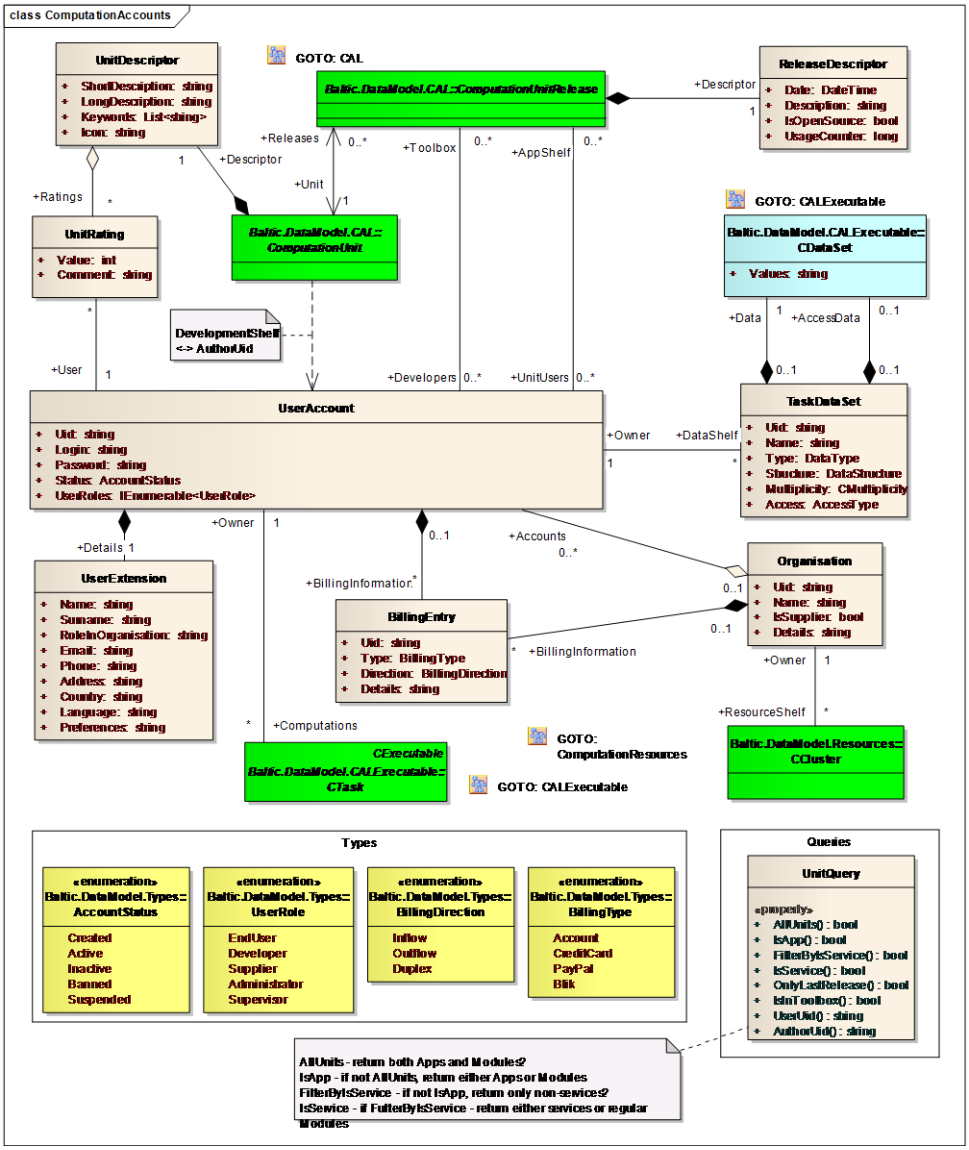

Figure 28, User Accounts

3.2 [O5.3] Computation Units and Computation Application Language (CAL)

These classes describe CAL and computation units which are being orchestrated using it. Detailed description of the language can be found in Chapter 5.

The central piece of the BalticLSC Software is computation unit (see `ComputationUnit` class). There are two types of computation units – computation applications (`ComputationApplication` class) and computation modules (`ComputationModule` class). Computation module is the atomic element of the computation unit – it is a Docker image that perform a computation with defined input and output. Computation module can be run on appropriate computation resource within any BalticLSC computation cluster. Computation applications are composed of other computation units, both modules and applications.

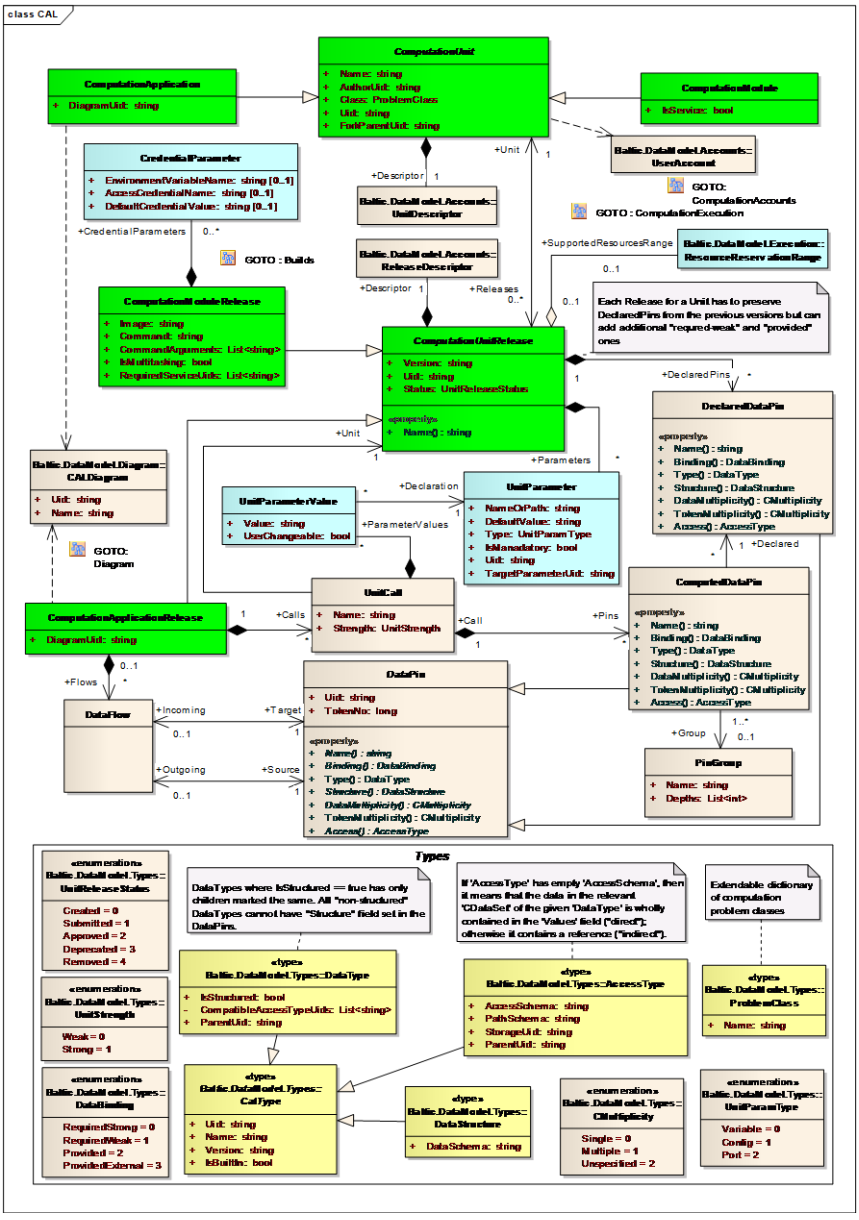

Figure 29, CAL Class Model (Metamodel)

The computation unit becomes usable in the moment when its first release is created (see class `ComputationUnitRelease`). Unit might have several releases, and, actually, computation applications use the particular release and not the unit itself. What an application release does, is described using CAL in a diagram (see class `CALDiagram`).

3.3 [O5.4] CAL Executable Language

CAL Executable language is an intermediate language which describes the process of execution of computation application on lower level of details – executable program. CAL Executable task contains information on jobs and batches containing jobs needed to perform the computation task. CAL operates on higher level of abstraction hiding these complex details from the user.

Thus, the `CTask` class describes the computation task (it conforms to the particular computation application release). Task consists of batches (class `CJobBatch`) which in turn consist of jobs (class `CJob`) and needed services (class `CService`). Jobs conforms to the particular computation module and batches are needed to group them into job groups to be executed on the single computation pod.

Inputs and outputs – data tokens - are described by `CDataToken` class.

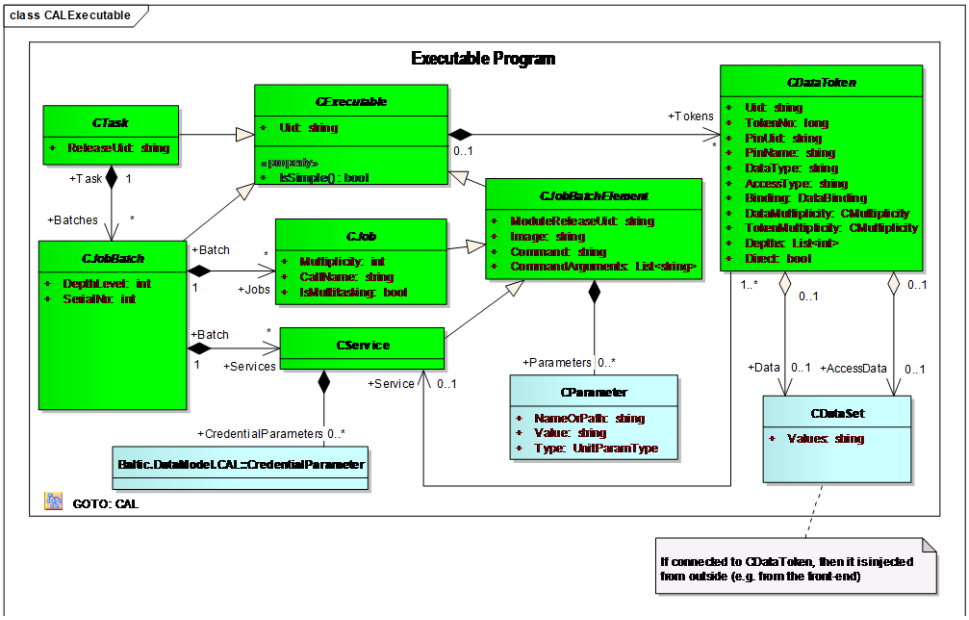

Figure 30, CAL Executable Language Class Model

3.4 [O5.4] CAL Messages

CAL Messages classes describe the messages used to communicate between computation batches, jobs, batch instances and job instances. It includes also passing token messages. Message queues are used for this purpose.

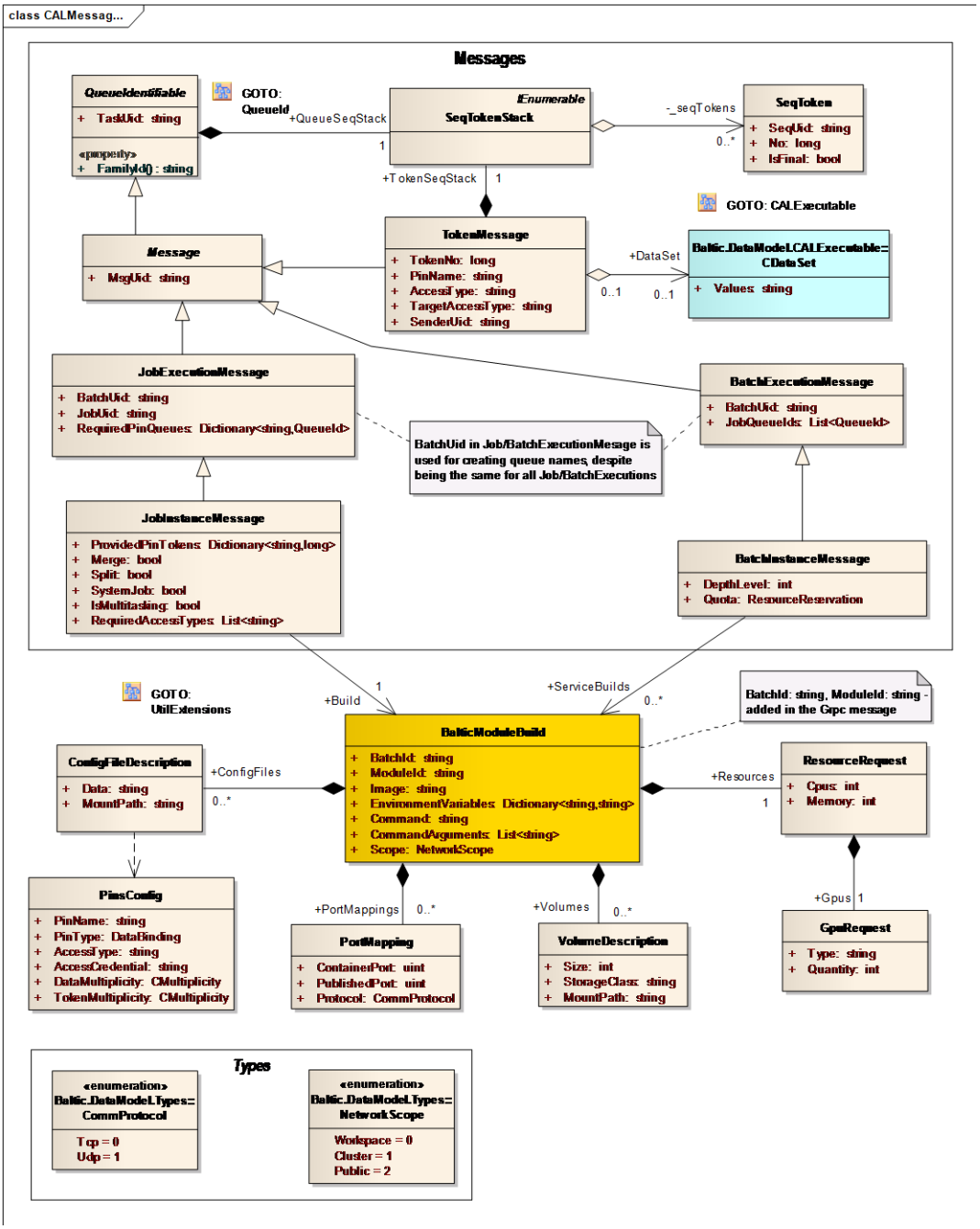

Figure 31, CAL Messages and Queues

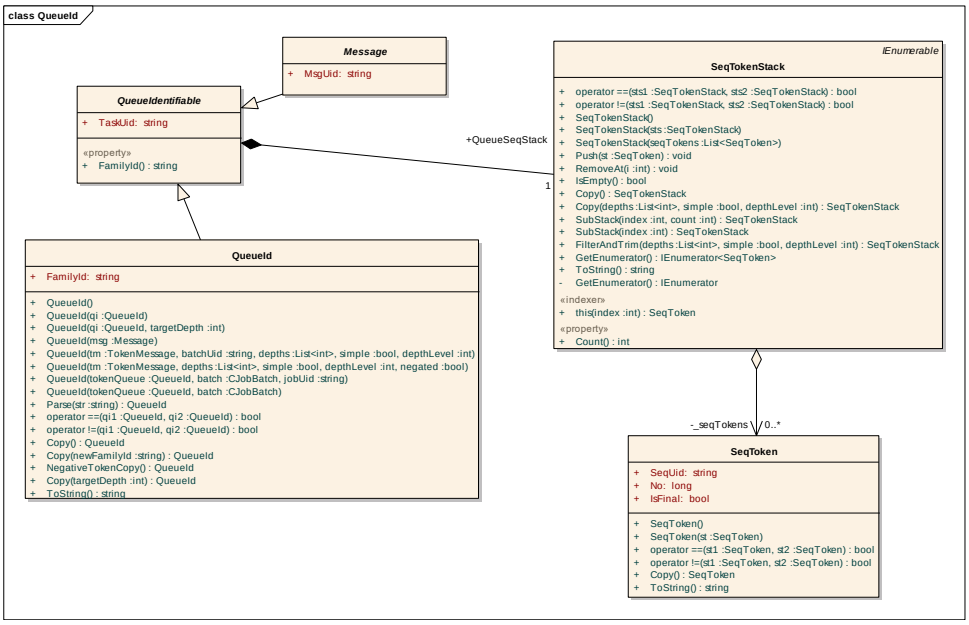

Figure 32, QueueId

3.5 [O5.4] Diagram

These classes describes the graphical diagrams (see class `CALDiagram`) and its elements used to depict the CAL programs. The main elements of diagrams are ports (class `Port`), boxes (class `Box`), and Lines (class `Line`). All elements might contain also compartments (class `Compartment`) – textual elements.

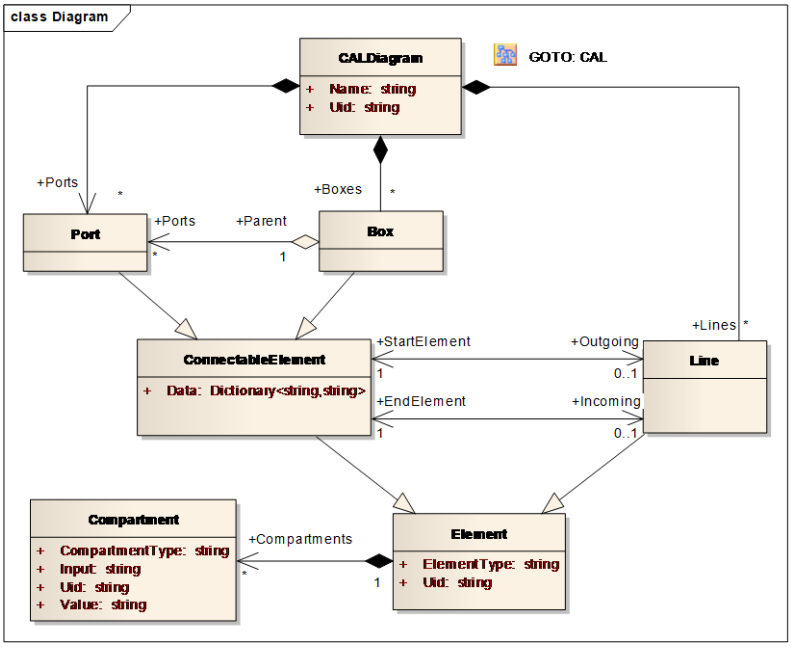

Figure 33, CAL Diagram Class Model

3.6 [O5.4] CAL Execution

CAL Execution classes hold execution records of the CAL programs. Every task, batch and job can be traced to its execution record – `TaskExecution`, `BatchExecution` and `JobExecution` class instances contain information on actual execution.

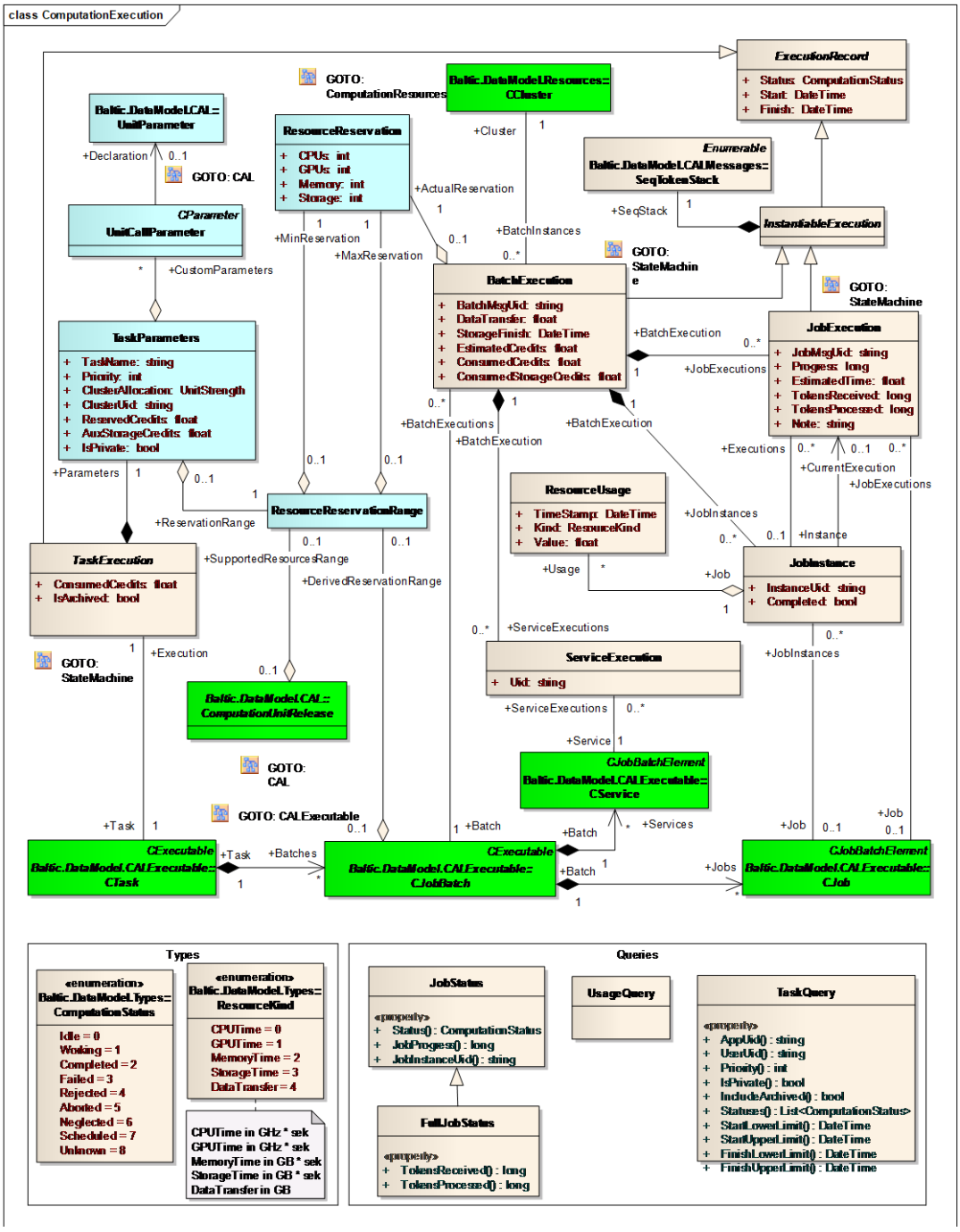

Figure 34, CAL Execution Class Model

3.7 [O5.2] Resources

Resources in the BalticLSC Network are provided as particular machines (class `CMachine`) within computation clusters (class `CCluster`).

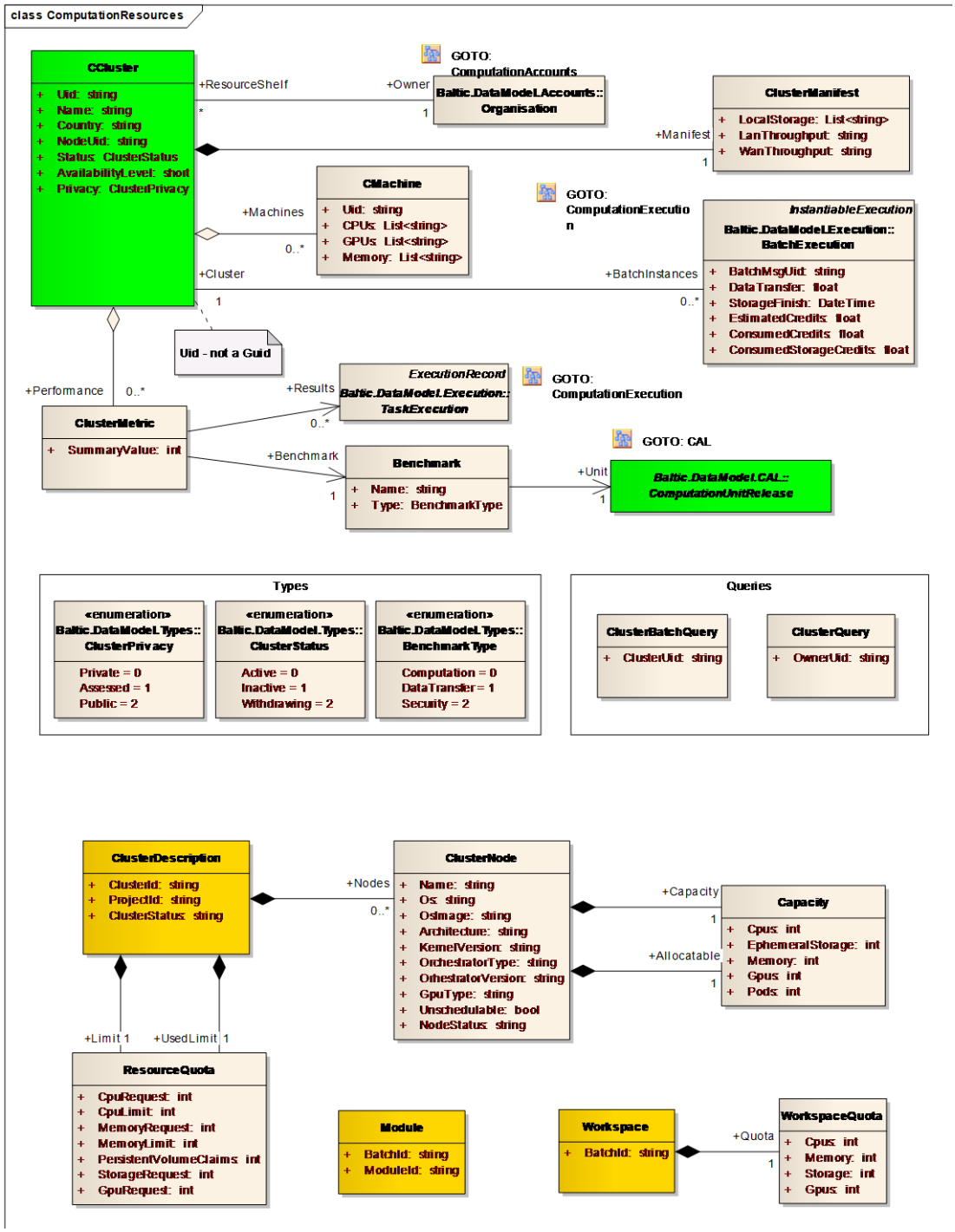

Figure 35, Network Resources Class Model

4. Behaviour Model

The behaviour model contains the detailed description of BalticLSC Software functions. The UML Use Case diagrams are used to list the functions and UML Sequence and UML Activity diagrams are used for detailed design of functions. They show which components are involved, what the sequence of calls to the API methods is and what the dataflow between components is.

4.1 Actors

As it has already been identified in O3.3 BalticLSC Software User Requirements Specification the main BalticLSC Software users can be divided into four categories: end-user, supplier (resource provider), developer and administrator:

Name	**End-user (App user)**
Description	End-users are using BalticLSC Environment to perform large scale computing with provided ready-to-use applications they can find in the BalticLSC Appstore. They are paying for used computing resources and applications with credits purchased at BalticLSC Environment. End-users use the functionality described in Section 4.2, Computation Application Usage.
Name	**Supplier (Resource provider)**
Description	Suppliers provide computing resources to the BalticLSC Network. In exchange, they receive credits from end-users for computations performed on provided resources. Suppliers use the functionality described in Section 4.4, Computation Resource Management.
Name	**Developer**
Description	Developers provide applications and modules to the BalticLSC AppStore. In exchange for use of their applications and modules, they receive credits from the end-users. Developers use the functionality described in Section 4.3, Computation Application Development.
Name	**Administrator**
Description	Administrators manage computing resources within the BalticLSC Network. Administrators are also responsible for approving new resource providers, new modules, and applications for the BalticLSC Environment. Administrators use functionality described in Sections 4.3 Computation Application Development and 4.5 Computation Application Supervision.

4.2 Computation Application Usage

This section describes the functionality (see Figure 36) used by the end-users (app users). It includes functionality to browse computation applications in the App-Store, add applications to computation cockpit, run computation tasks, and manage user owned datasets.

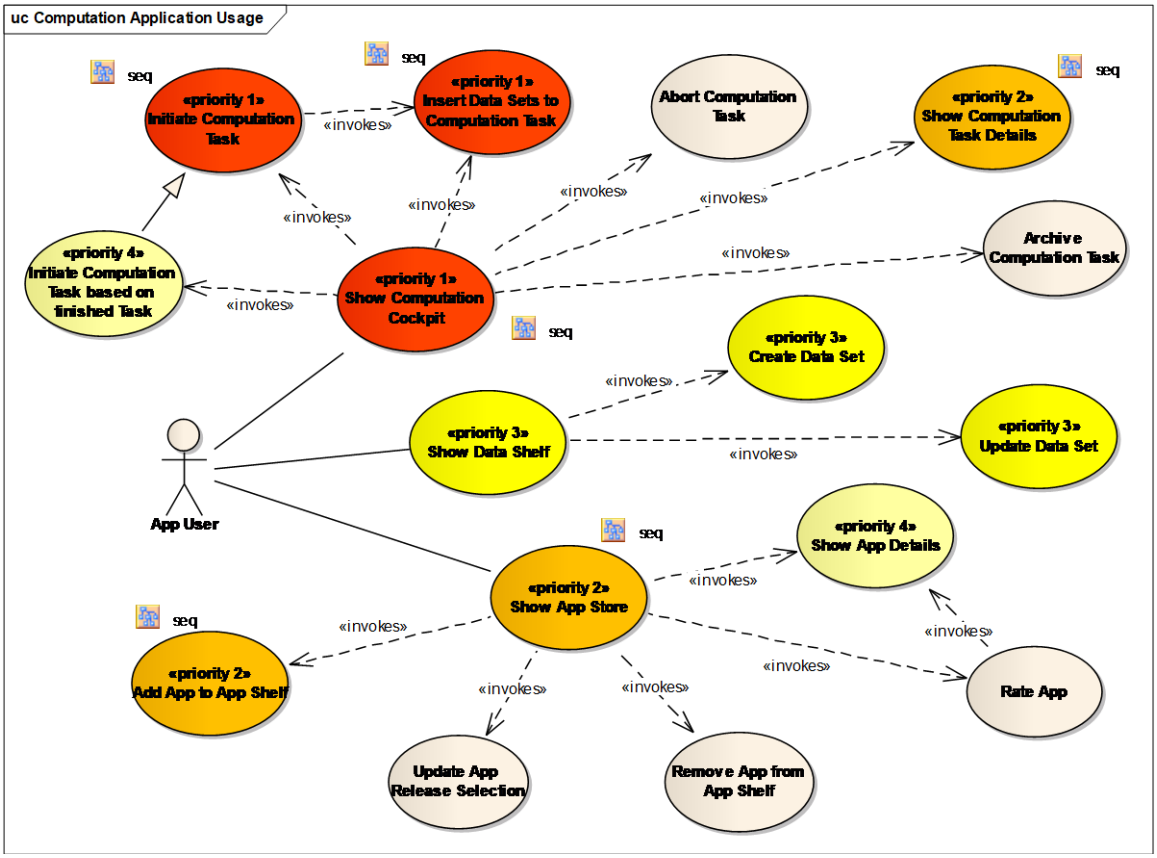

Figure 36, Computation Application Usage

End-users use three main groups of functions and there are three main components in the Front-end for that:

1. Computation Cockpit – runs computation tasks.
    a. Initiate Computation Task
    b. Insert Data Sets to Computation Task
    c. Show Computation Cockpit
    d. Show Computation Task Details
    e. Archive Computation Task
    f. Abort Computation Task
2. App Store – allows access to apps.
    a. Show App Store
    b. Add App to App Shelf
    c. Show App Details
    d. Update App Release Selection
    e. Remove App from App Shelf
3. [O5.2] Data Shelf – allows to register user's datasets.
    a. Show Data Shelf
    b. Create Data Set
    c. Update Data Set

4.2.1 [O5.4] Computation Task Execution

This Section contains the design of task execution.

Design of main user interactions is depicted in Figure 37, Figure 38, Figure 39, Figure 40

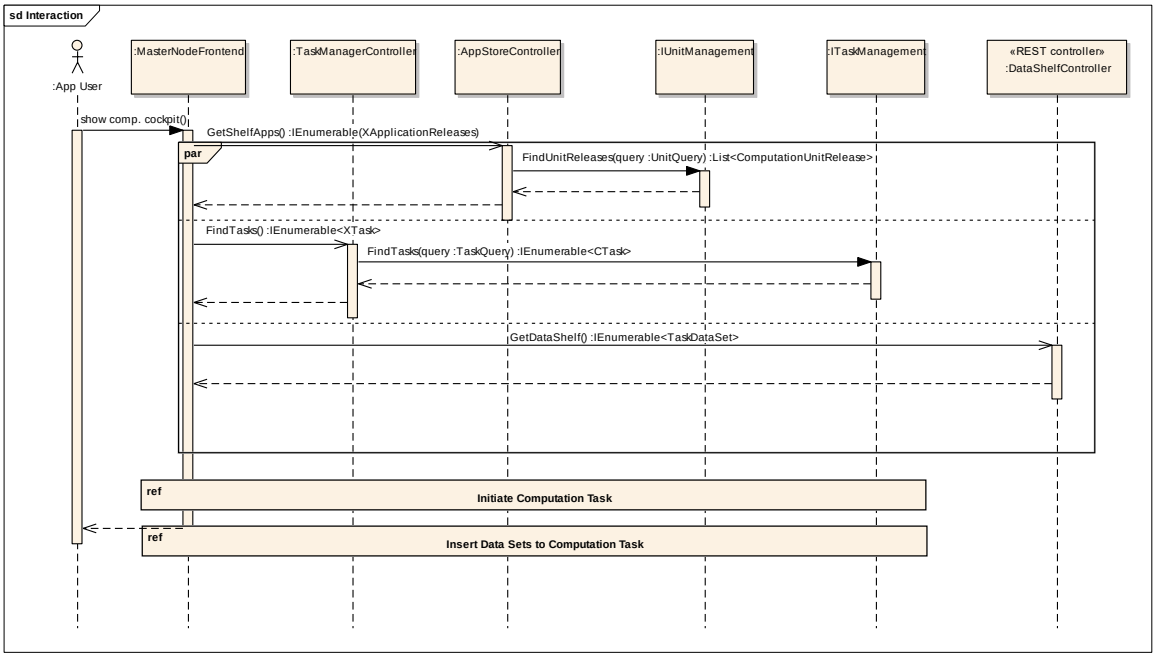

Figure 37, User Interaction - Show Computation Cockpit

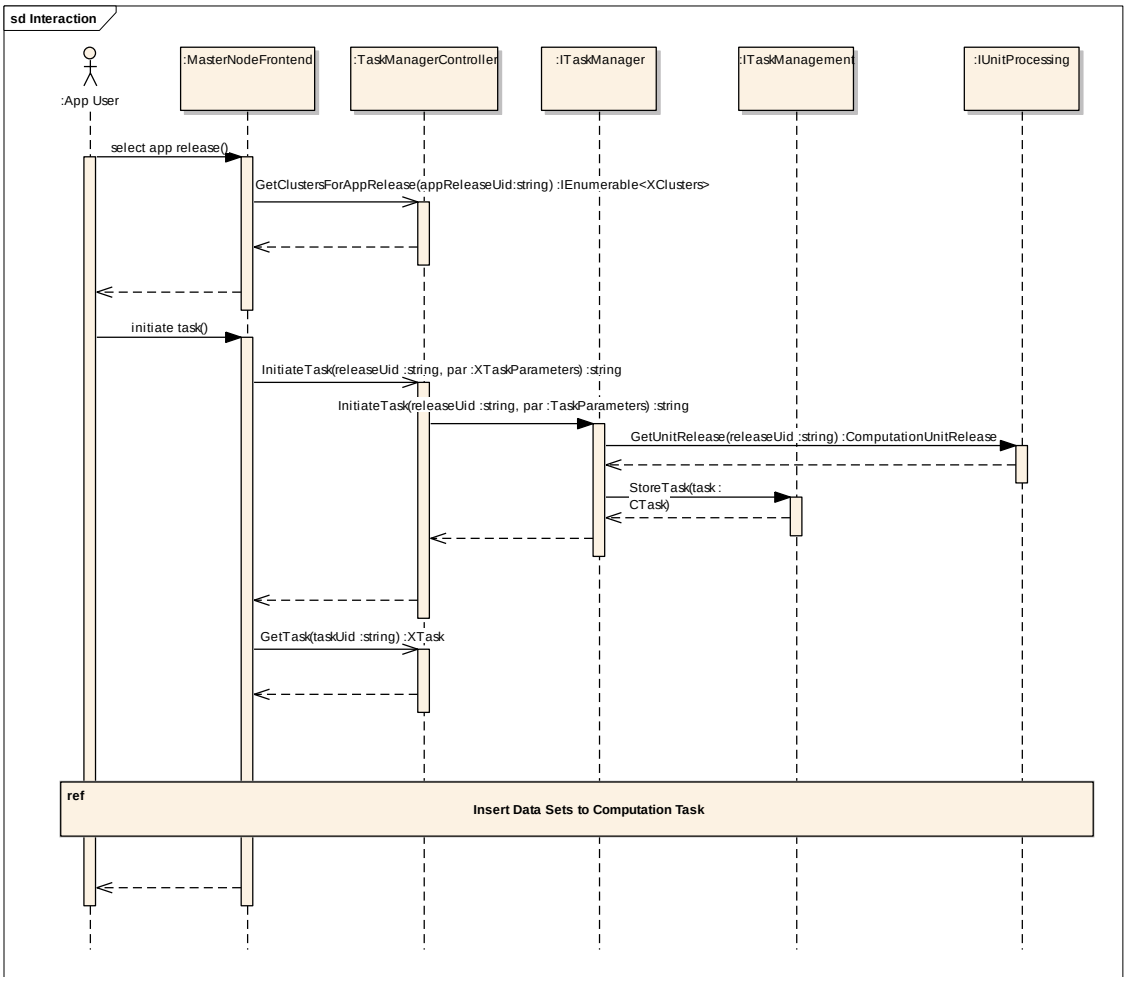

Figure 38, User Interaction - Initiate Task

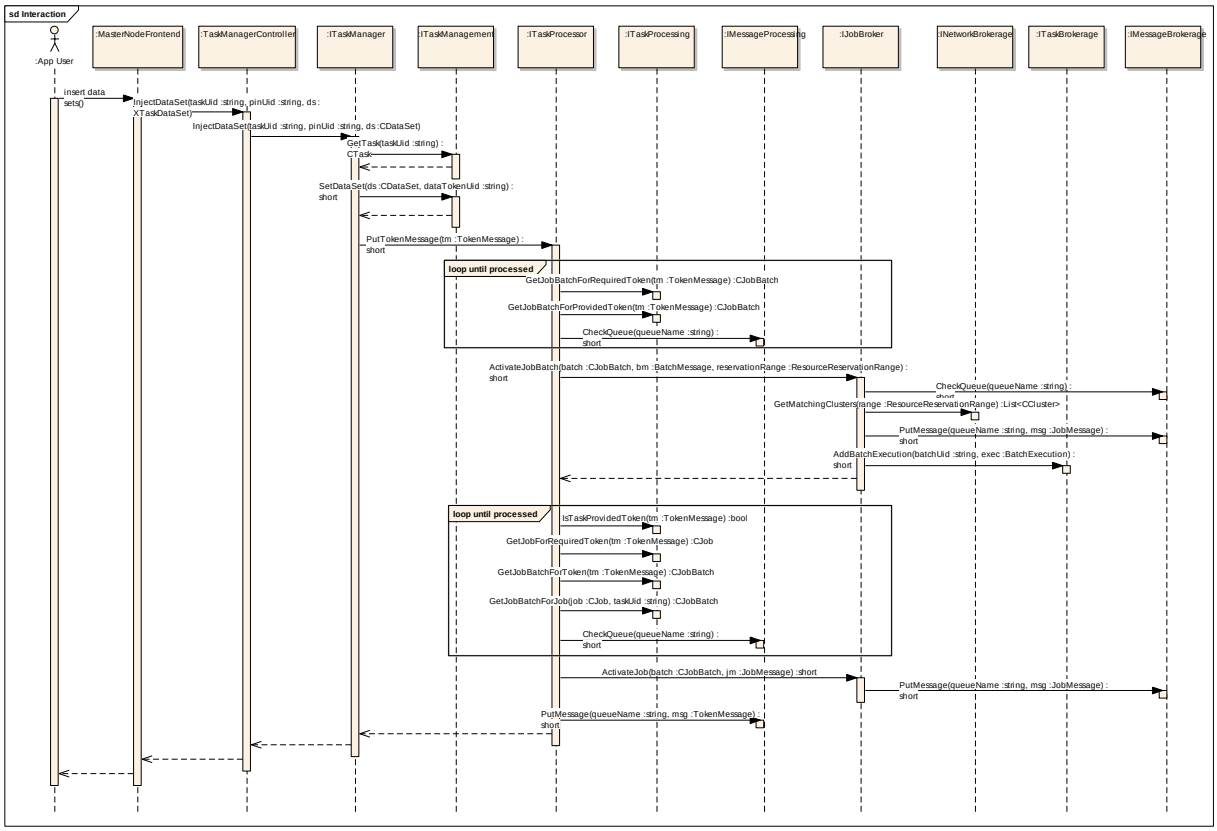

Figure 39, User Interaction - Insert Data Sets

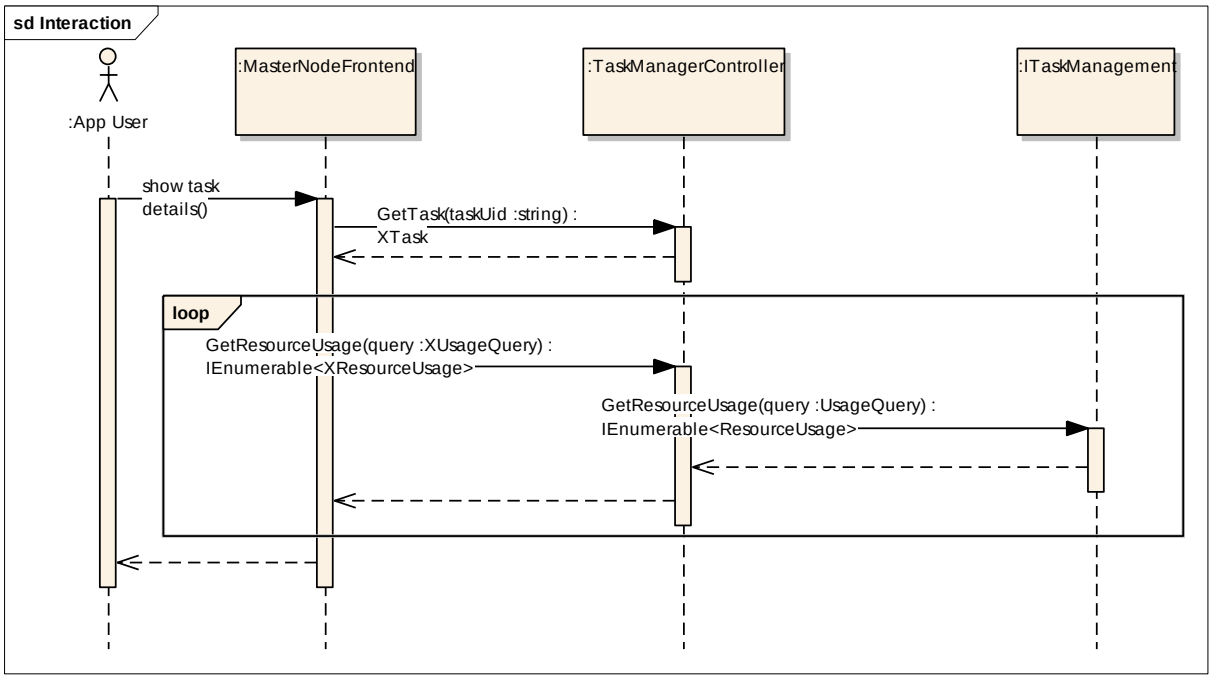

Figure 40, User Interaction - Show Task Details

Design of the main functions needed to actually execute the computation task and performed by BalticLSC Software is depicted in Figure 41, Figure 42, Figure 43, Figure 44, Figure 45, Figure 46. We use UML Sequence diagrams to show the sequence of operations performed by BalticLSC Software components.

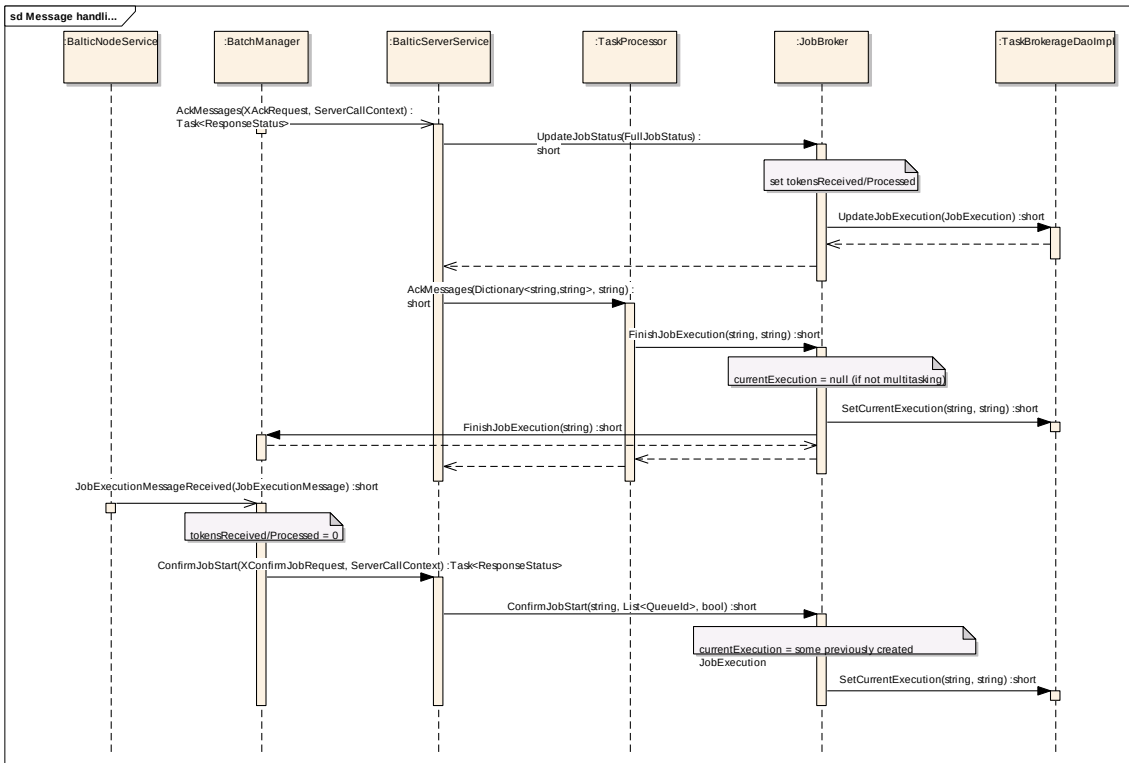

Figure 41, Handling Job Execution Messages

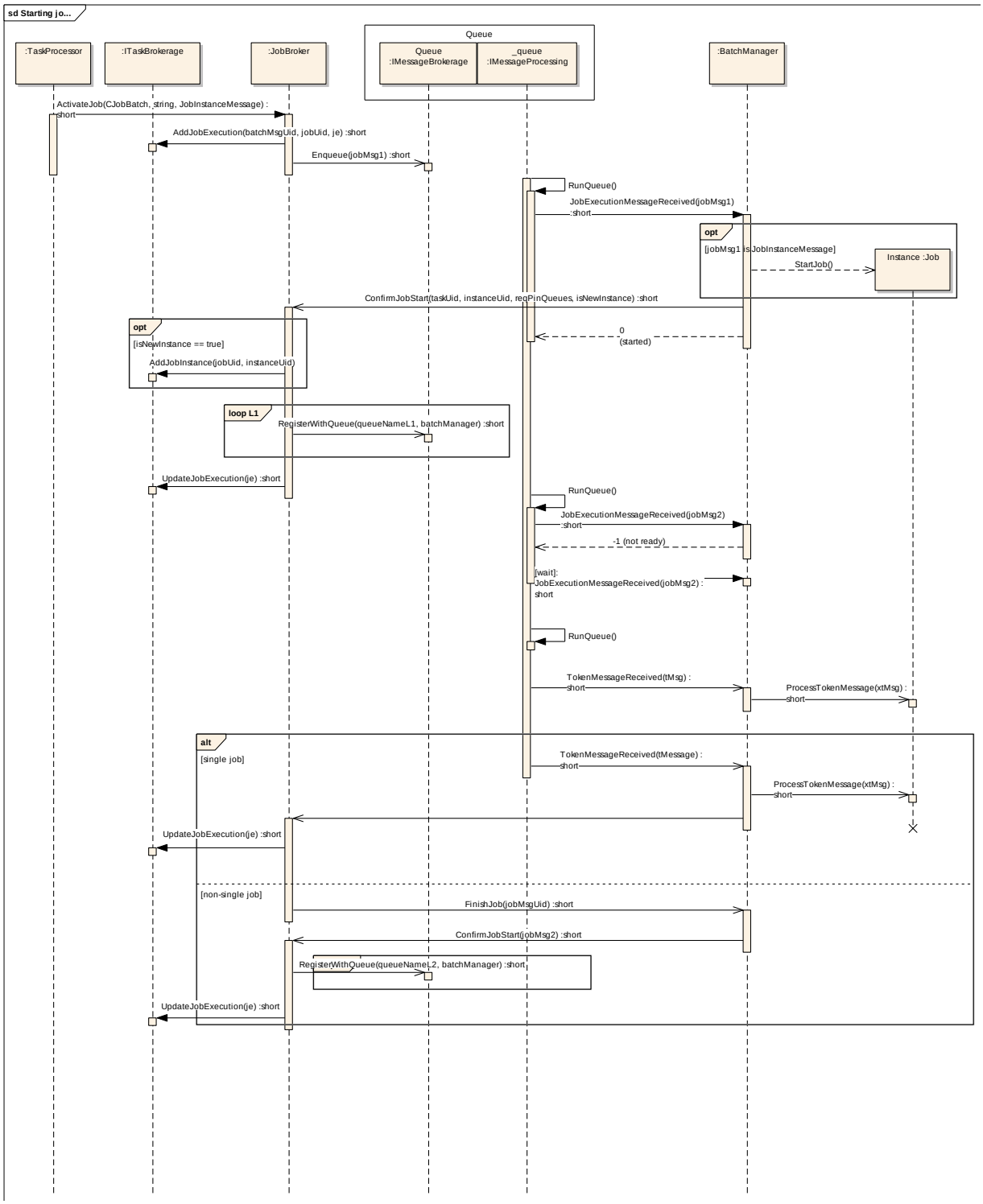

Figure 42, Starting Job

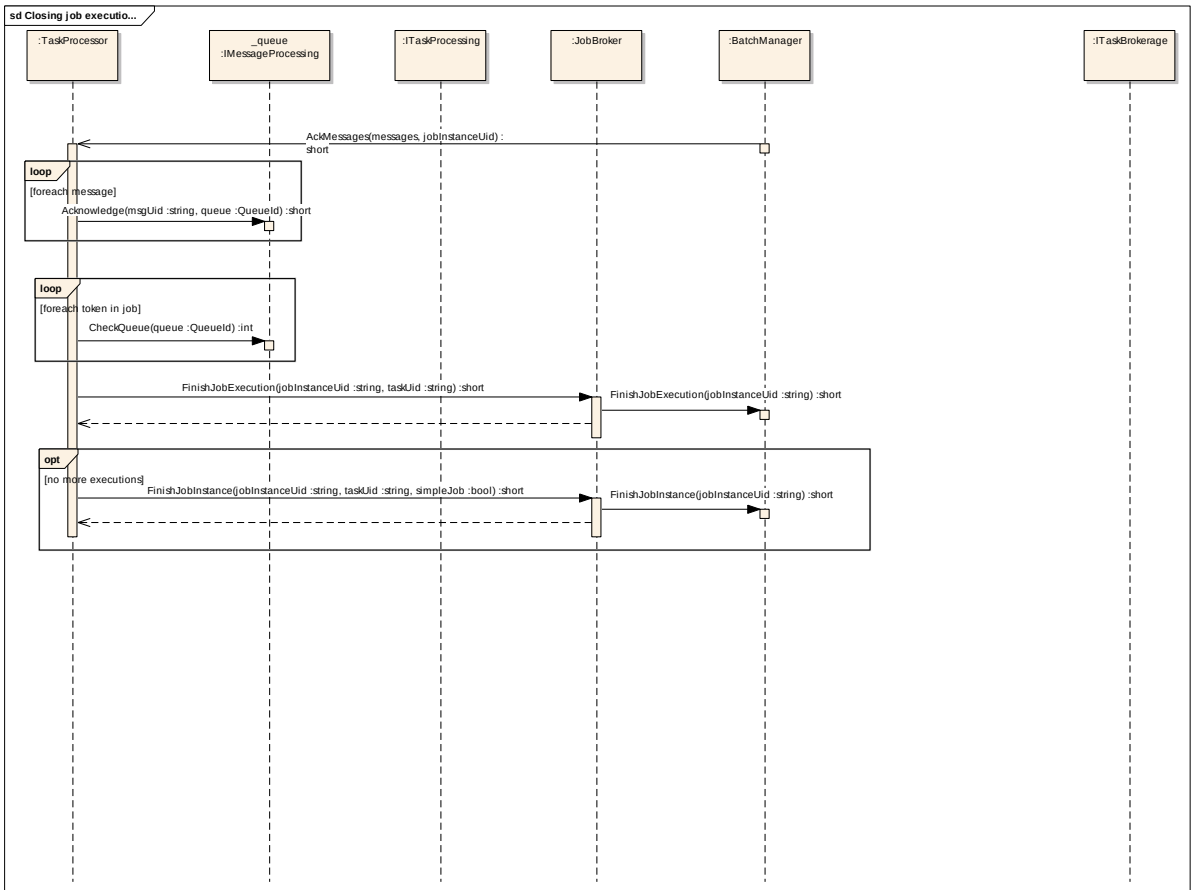

Figure 43. Closing Job

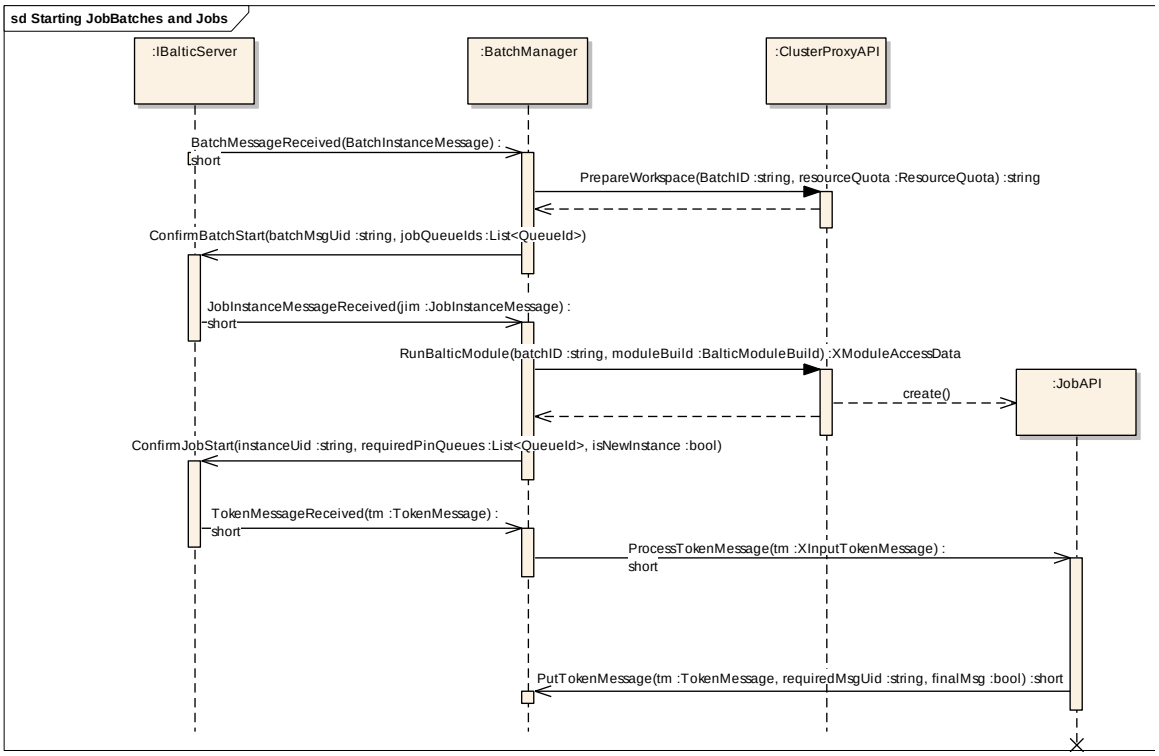

Figure 44, Token Circulation - Starting Job Batches

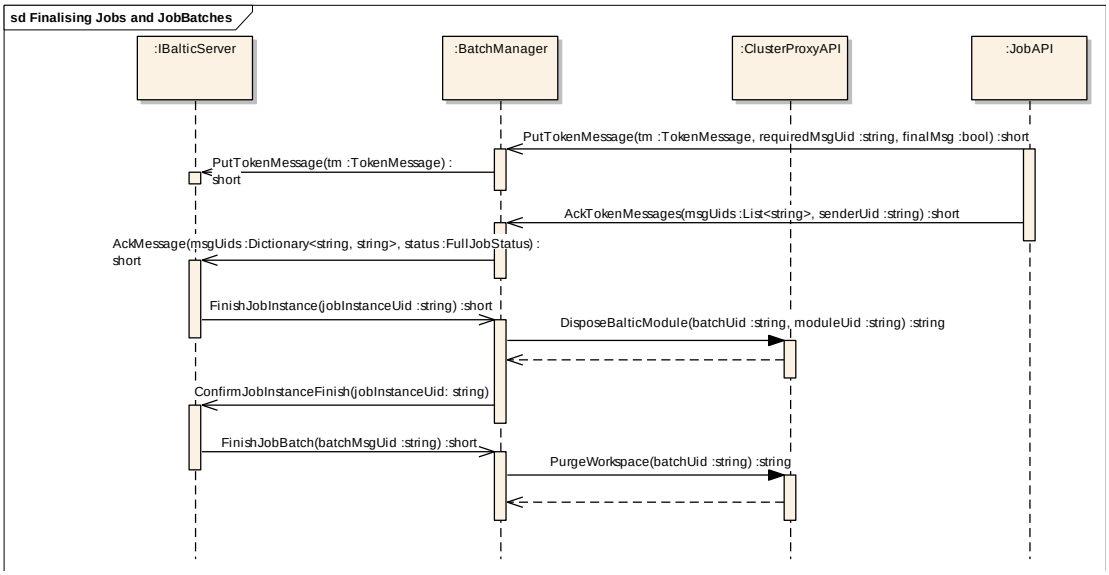

Figure 45, Token Circulation - Finalising Jobs

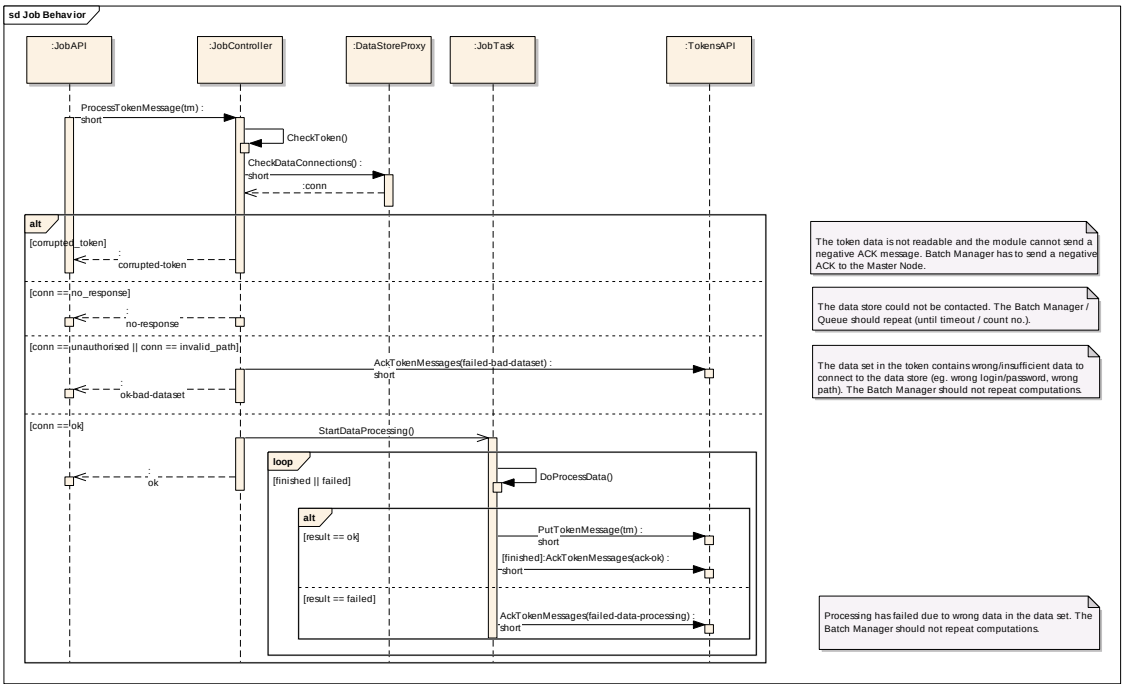

Figure 46, Job Behavior

Next, we provide UML Activity diagrams which explain in detail how the central activity of the BalticLSC Environment is performed – starting and closing jobs (computations). See Figure 47, Figure 48, Figure 49.

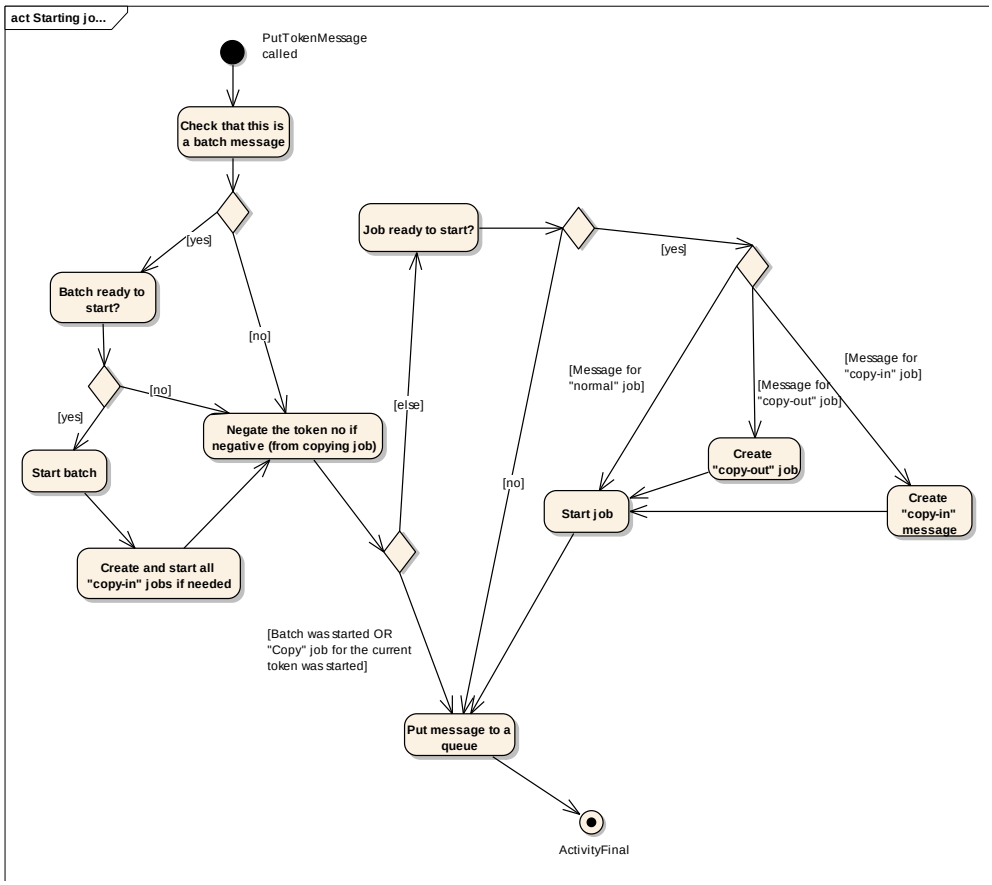

Figure 47, Starting Job

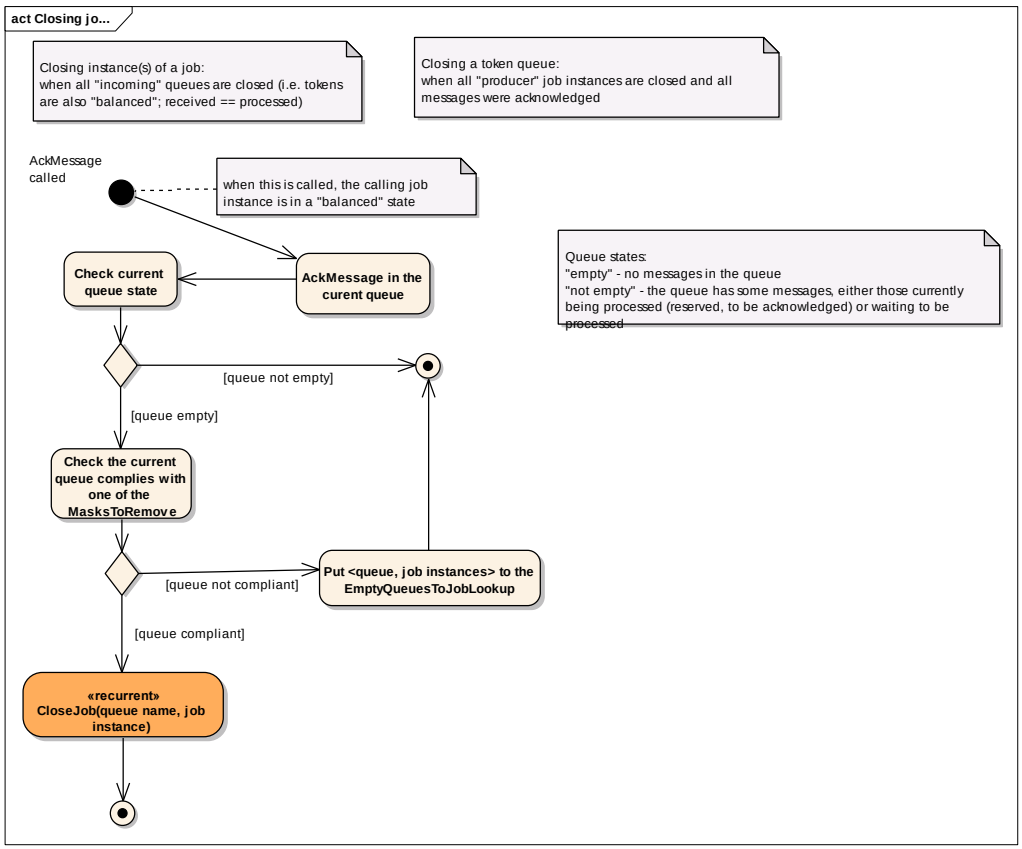

Figure 48, Closing Jobs

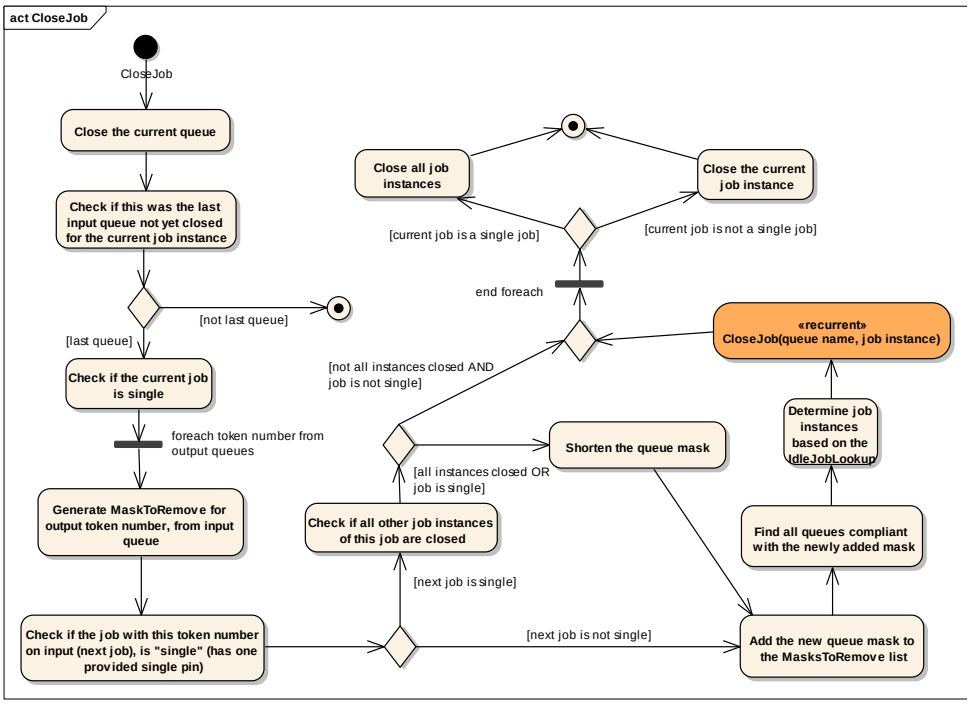

Figure 49, Closing Job

Next, we add UML State Machine diagrams depicting how states of tasks, batches and jobs changes over time. See Figure 50, Figure 51, Figure 52.

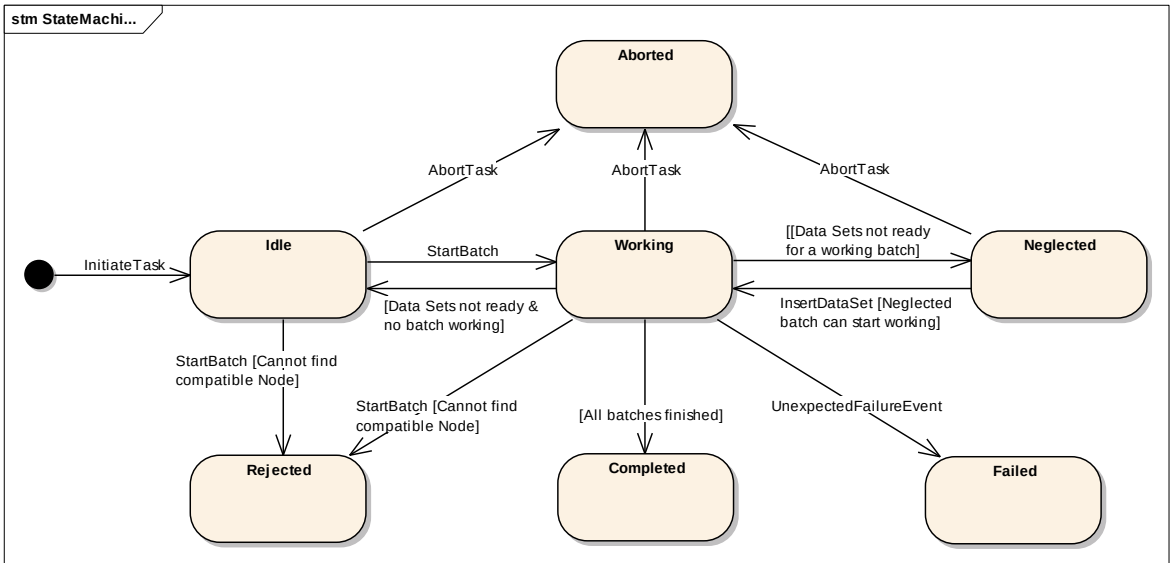

Figure 50, Task Status State Diagram

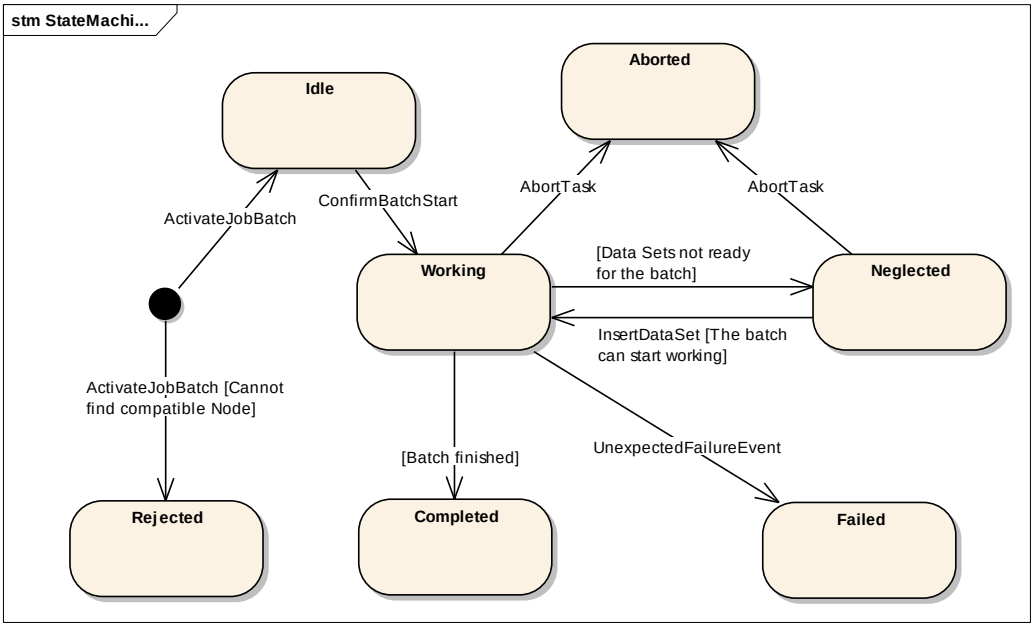

Figure 51, Batch Status State Diagram

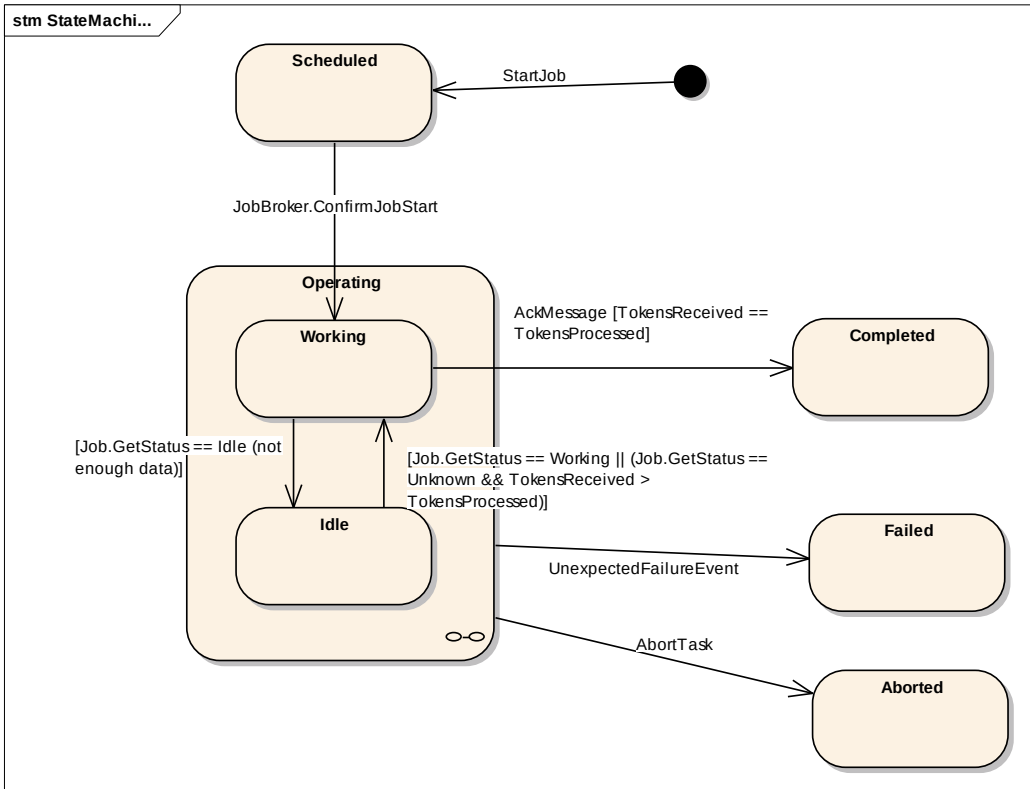

Figure 52, Job Status State Diagram

4.2.2 [O5.2] App Store Functions

This Section contains the design of relatively simple app store functions.

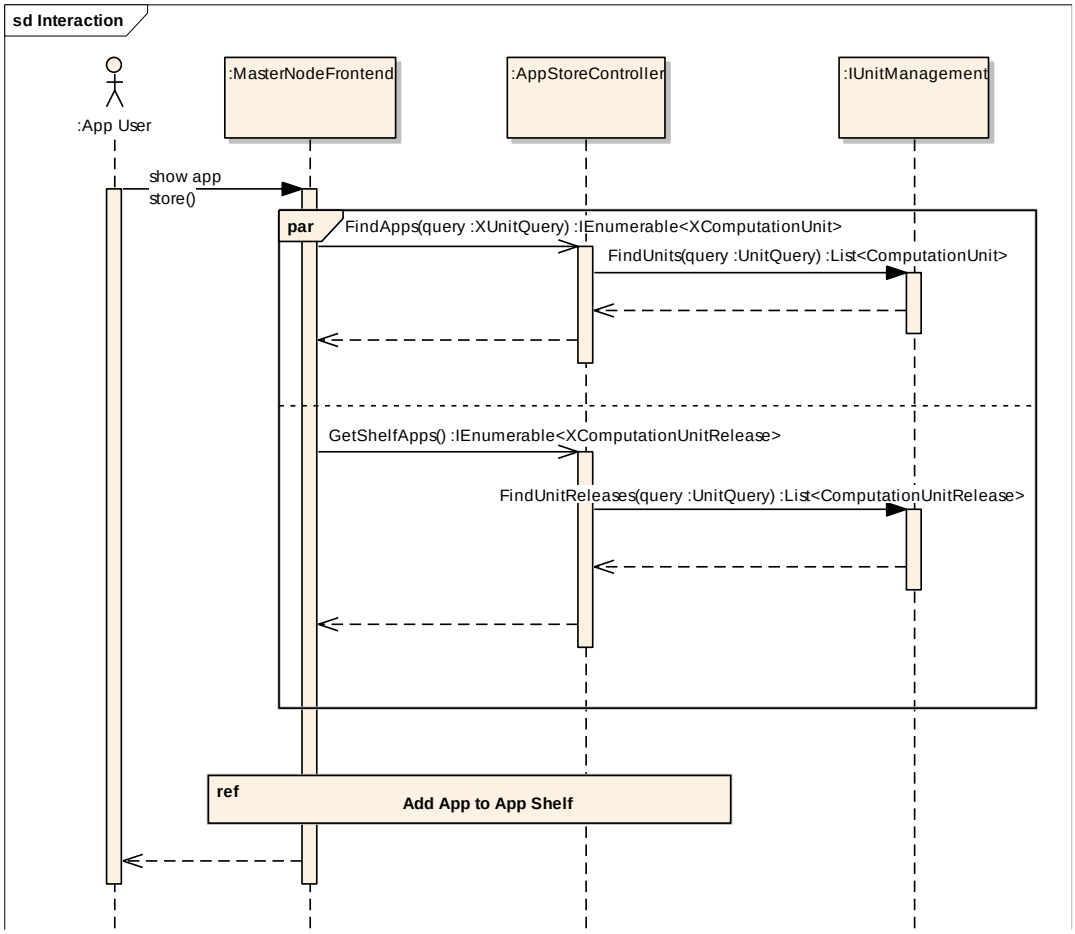

Figure 53, User Interaction - Show App Store

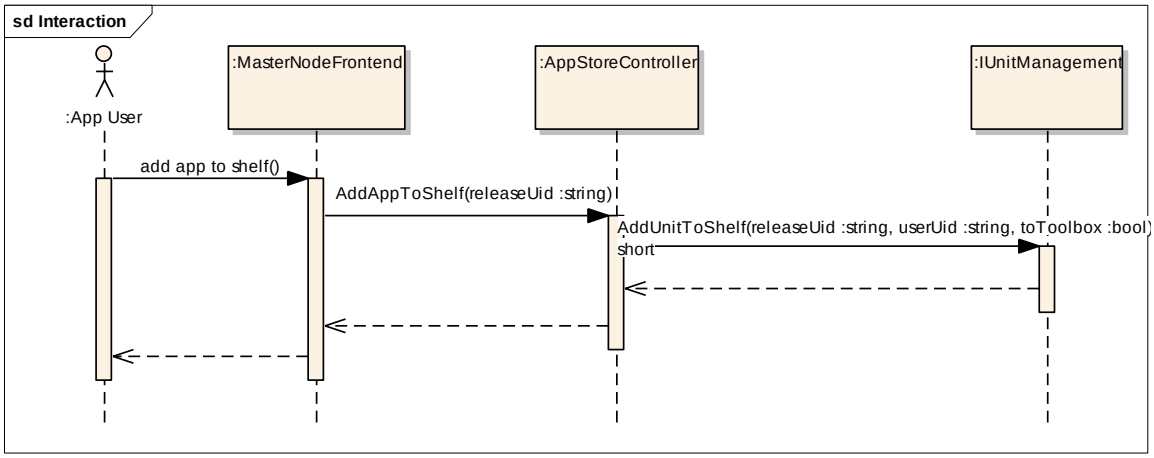

Figure 54, User Interaction - Add App Too App Shelf

4.3 [O5.2] [O5.4] Computation Application Development

This section describes the functionality (see Figure 55Figure 36) used by the developers. It includes functionality to create computation applications out of readymade modules, and to create modules itself.

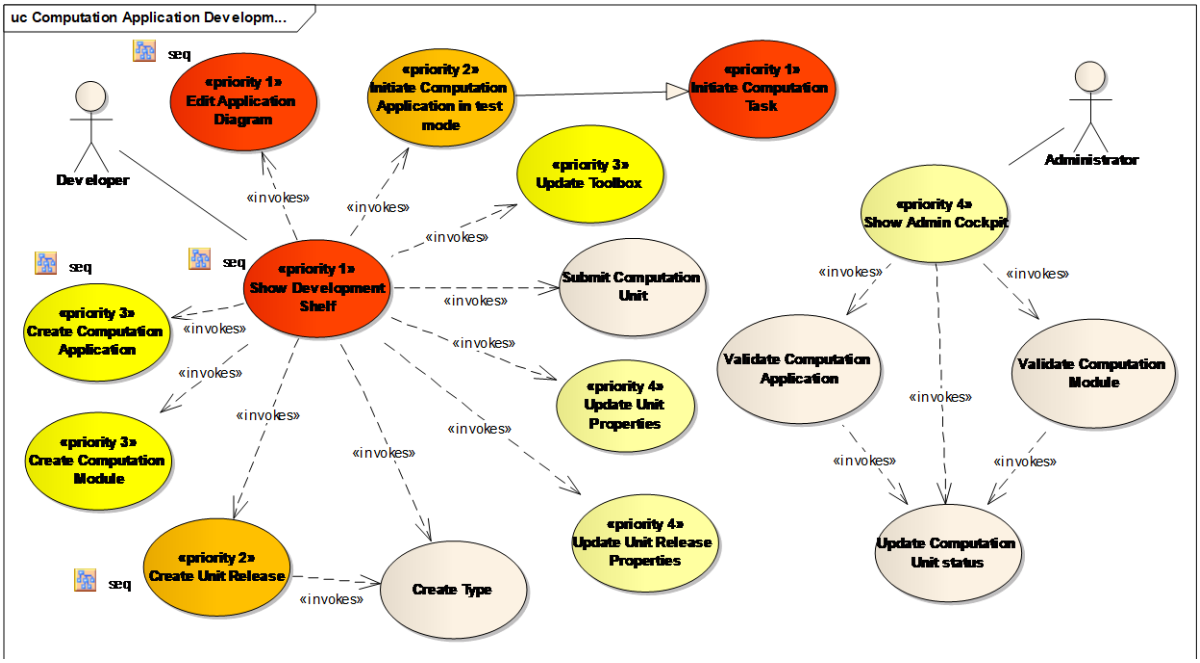

Figure 55, Computation Application Development

Developer use such main functions:

1. [O5.2] Show Development Shelf – manages own computation units and browse readymade units of other developers.
2. [O5.2] Create Computation Application and Create Computation Module – creates computation units.
3. [O5.4] Edit Application Diagram – builds CAL programs.
4. [O5.2] Create Unit Release – creates computation unit releases.
5. [O5.2] Update Unit Properties and Update Unit Release Properties – edits unit properties.
6. [O5.4] Update Toolbox – manages toolbox.

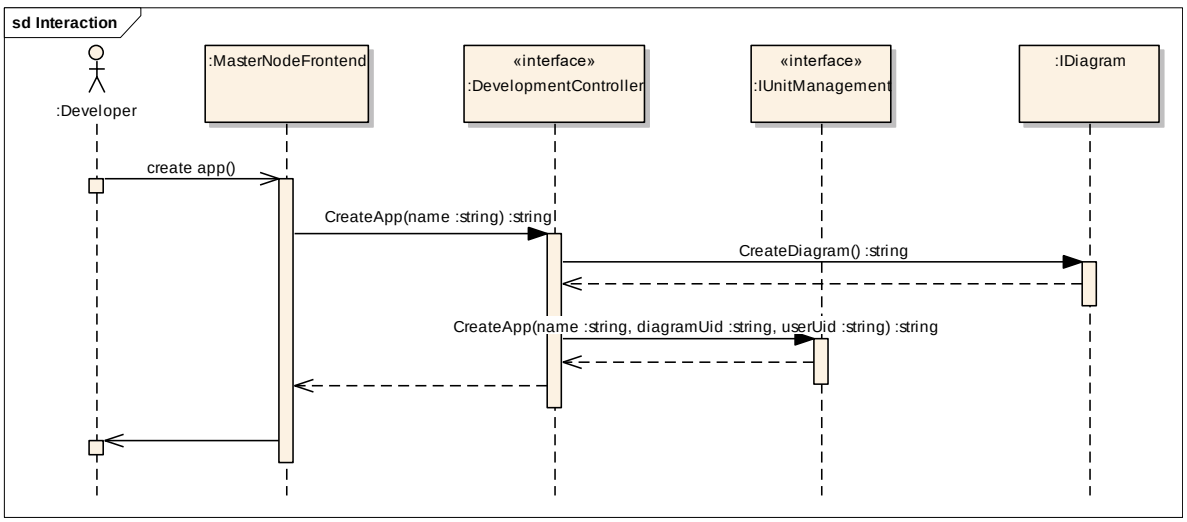

Figure 56, User Interaction - Create Computation Application

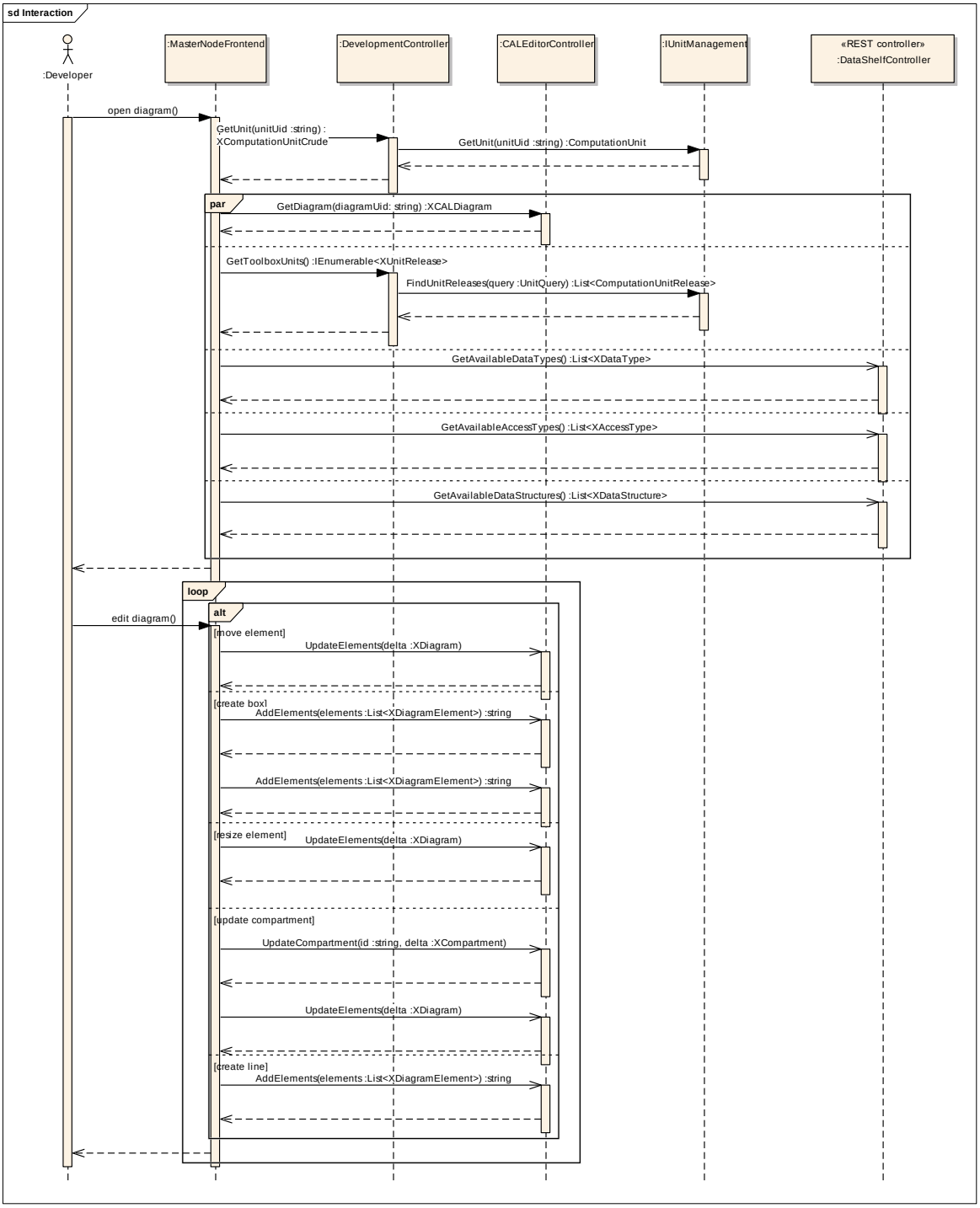

Figure 57, User Interaction - Edit Application Diagram

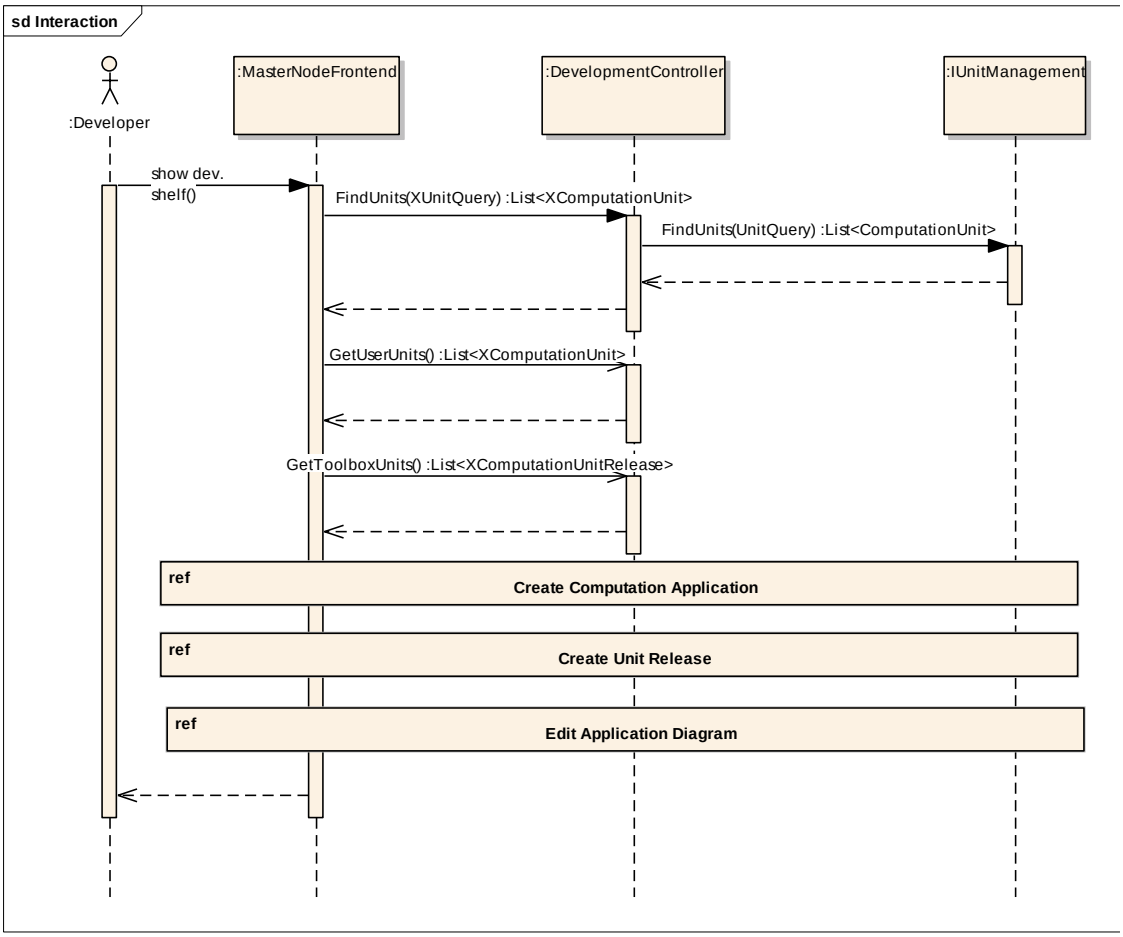

Figure 58, User Interaction - Show Development Shelf

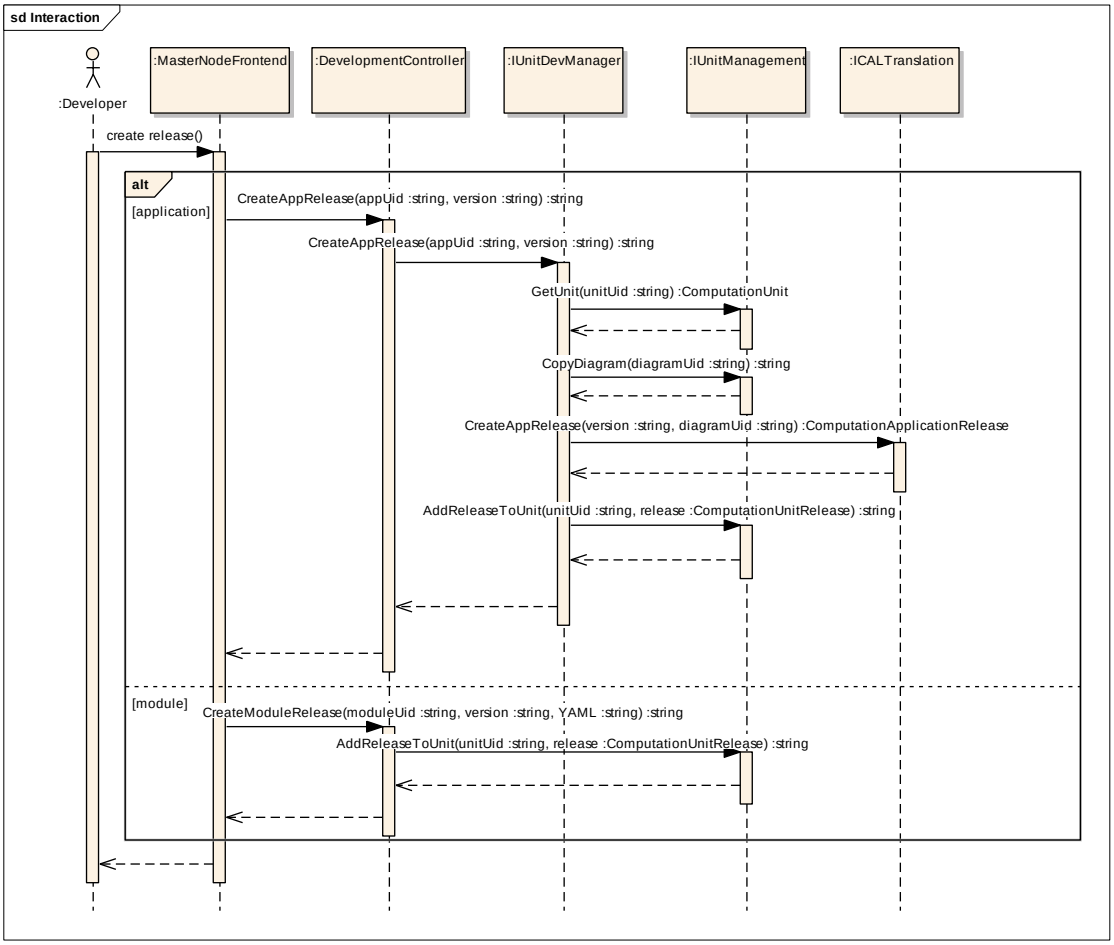

Figure 59, User Interaction - Create Unit Release

4.4 Computation Resource Management

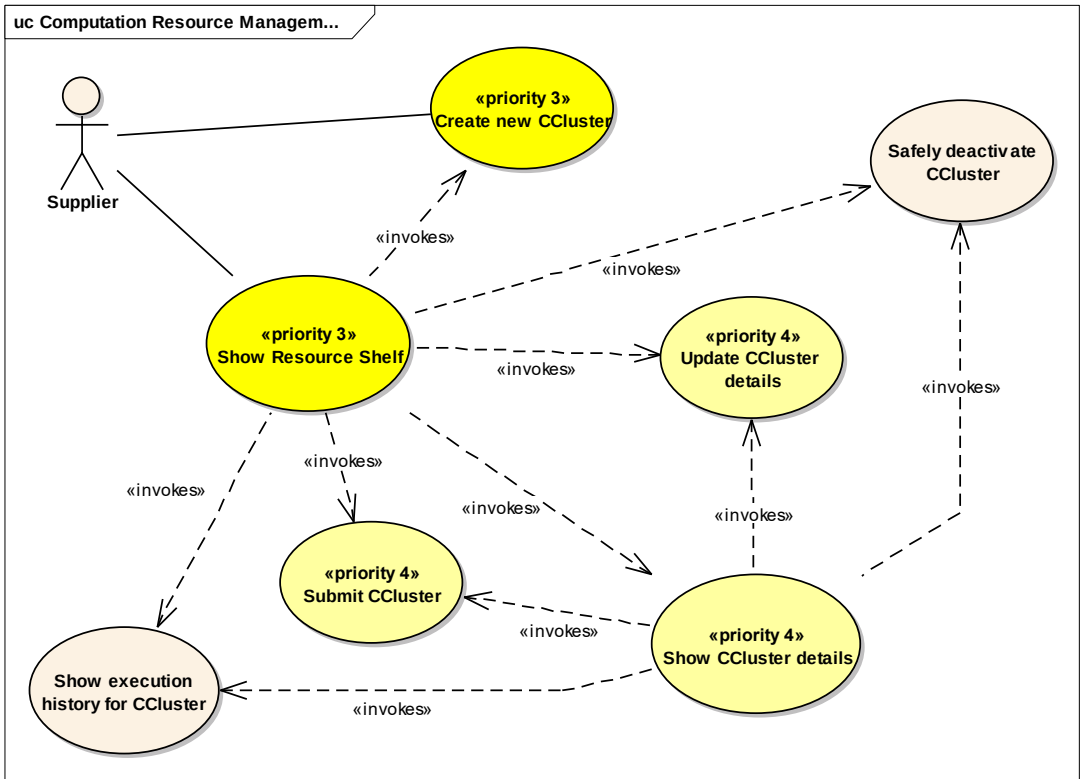

Figure 60, Computation Resource Management

4.5 Computation Resource Supervision

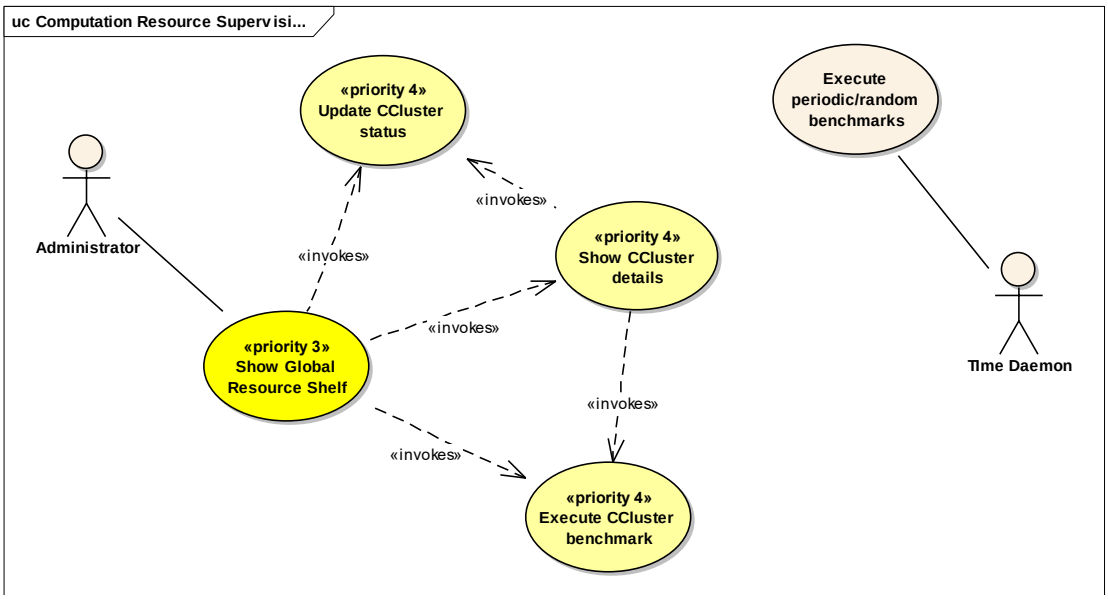

Figure 61, Computation Resource Supervision

5. [O5.3] Computation Application Language (CAL)

This Chapter describes the Computation Application Language (CAL) - main concepts, abstract and concrete syntaxes, and semantics. The CAL metamodel (See Figure 62, CAL Metamodel as UML Class diagram is used to denote main classes of CAL and its surroundings. It corresponds to the Output 5.3B.

5.1 Surroundings

Computations within BalticLSC Network is being caried out by computation units (class `ComputationUnit`). There are two different kinds of computation units – applications (class `ComputationApplication`) and modules (class `ComputationModule`).

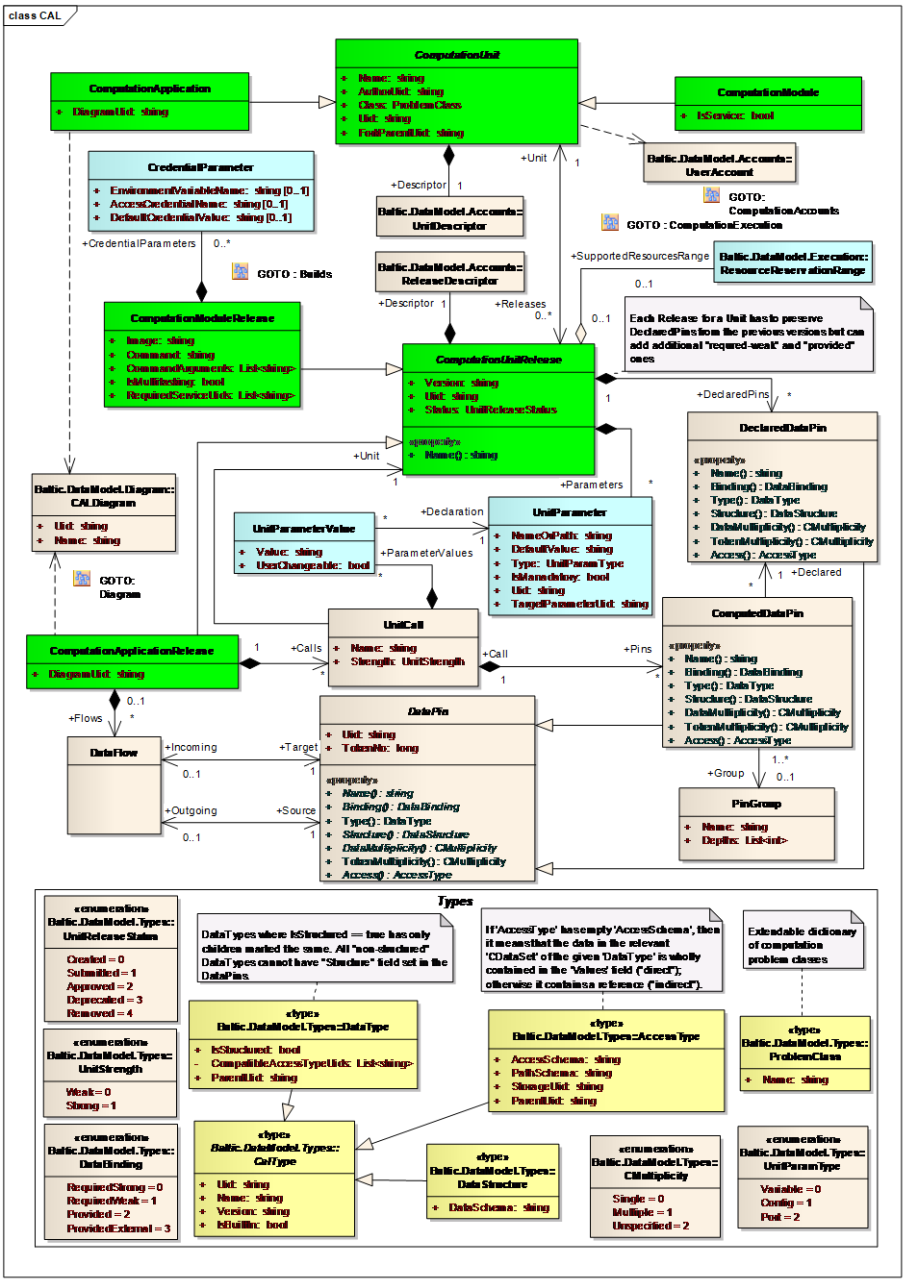

Figure 62, CAL Metamodel

Computation module is the atomic element of the computation – it is a Docker image which implements the `JobAPI` interface (see Figure 8, Job Controller) and communicates with the batch manager using `TokensAPI` interface. They are developed by the module developers and are the building blocks for computation applications. Basically, computation application consists of calls to the computation modules and other computation applications. CAL is used to denote the computation application in a graphical way, thus, every computation application has a diagram (class `CALDiagram`) attached to it (modules does not have).

Computation unit is useable for the CAL application if the unit has a release (class `ComputationUnitRelease`). Computation unit release is a fixed state of the computation unit which is executable within BalticLSC Environment. Computation unit may have several releases and every non-deprecated release is usable for construction of computation application.

5.2 CAL Program

CAL program describes the computation of a particular release of computation application. Thus, in abstract syntax we do not distinguish between CAL program and computation application release. In the metamodel CAL program corresponds to the `ComputationApplicationRelease` class.

5.2.1 `ComputationApplicationRelease` class

Brief overview. Computation application release contains all CAL elements: declared pins, unit calls and data flows. It corresponds to a single diagram in the concrete syntax of CAL.

Generalizations. It is specialized from `ComputationUnitRelease` class.

Attributes.

[*inh*] Name: string – specifies the name of the release.

[*inh*] Uid: string – specifies the GUID of the release.

[*inh*] Status: UnitReleaseStatus – specifies the status of the release. May be, Created=0, Submitted=1, Approved=2, Deprecated=3 and Removed=4.

[*inh*] Version: string – specifies the version of the release.

DiagramUid: string – specifies the GUID of the corresponding diagram.

Associations.

[*inh*] Descriptor [1] – additional (non-essential for the execution) information on the release.

[*inh*] Unit [1] – unit the release belongs to.

[*inh*] DeclaredPins [*] –list of declared data pins for the release.

[*inh*] SupportedResourcesRange [0..1] – the range of resources (CPU, RAM, etc. …) the release is capable to operate.

[*inh*] Parameters [*] – list of parameters for the release.

Calls [*] – list of unit calls owned by the release.

Flows [*] – list of data flows owned by the release.

Concrete Syntax. Computation application release is depicted as a diagram.

5.2.2 `ComputationUnitRelease` class

Brief overview. Computation unit release is an abstraction of both computation application and module releases. In the CAL program the calls are made to unit releases without distinction between modules and applications. The CAL program developer can even not know which kind of unit is used.

Attributes.

Name: string – specifies the name of the release.

Uid: string – specifies the GUID of the release.

Status: UnitReleaseStatus – specifies the status of the release. May be, Created=0, Submitted=1, Approved=2, Deprecated=3 and Removed=4.

Version: string – specifies the version of the release.

Associations.

Descriptor [1] – additional (non-essential for the execution) information on the release.

Unit [1] – unit the release belongs to.

DeclaredPins [*] –list of declared data pins for the release.

SupportedResourcesRange [0..1] – the range of resources (CPU, RAM, etc. …) the release is capable to operate.

Parameters [*] – list of parameters for the release.

5.2.3 `DeclaredDataPin` class

Brief overview. Declared data pins declare datasets required and provided by computation unit. Data pin defines data type (e.g., JSON, XML, Image, etc. ...), structure (for structured data types), and access type (e.g., MongoDB, FTP) of the dataset. Data multiplicity can be defined too – it means that dataset might consist of single item or it might be a list of items. Token multiplicity defines whether a data pin requires or provides single or multiple tokens. Declared data pins of the computation application are part of the CAL program (while declared pins of the modules are not).

Generalizations. It is specialized from `DataPin` class.

Attributes.

[*inh*] Name: string – specifies the name of the data pin.

[*inh*] Uid: string – specifies the GUID of the data pin.

[*inh*] TokenNo: long – specifies a token number.

[*inh*] Binding: DataBinding – specifies the data binding for the data pin. Binding might be RequiredStrong=0, RequiredWeak=1, Provided=2, ProvidedExternal. Required data pins defines datasets that are consumed, provided – produced. Strong data pins define mandatory datasets, weak – optional.

[*inh*] Type: DataType – specifies type of the dataset, e.g., JSON, XML, Image, etc.

[*inh*] Structure: DataStructure – specifies the data structure, in fact, data schema of the dataset, if the data type is structured.

[*inh*] DataMultiplicity: CMultiplicity – specifies the multiplicity of data items in the dataset. It might be Single=0, Multiple=1.

[*inh*] TokenMultiplicity: CMultiplicity – specifies the multiplicity of tokens consumed or produced by the data pin. It might be Single=0, Multiple=1.

[*inh*] Access: AccessType – specifies way to access dataset, e.g., MongoDB, FTP, etc. …

Associations.

[*inh*] Incoming [0..1] – incoming data flow. Required declared data pins (Binding = RequiredWeak or RequiredStrong) may not have incoming data flows.

[*inh*] Outgoing [0..1] – outgoing data flow. Provided declared data pins (Binding = Provided) may not have outgoing data flows.

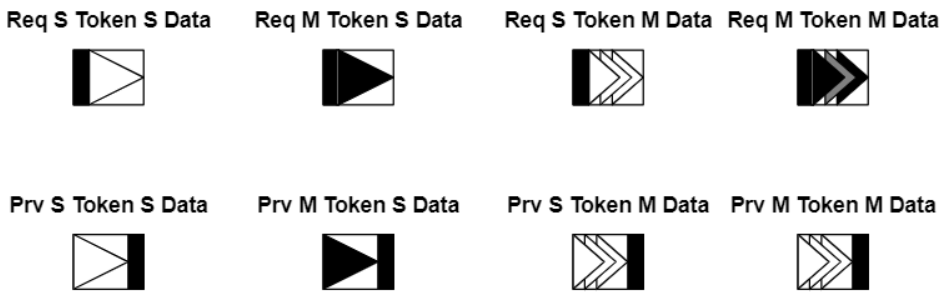

Figure 63, Notation of declared pins depending on their properties. Req - required, S - single, M - multiple, Token - token multiplicity, Data - data multiplicity.

Concrete Syntax. Declared data pins are defined differently for computation modules and computation applications. Pins of the modules are defined using conventional GUI within the BalticLSC Computation Tool. However, declared pins for applications are defined within CAL programs using graphical notation – rectangle. The shape of the rectangle is determined by attribute values. The upper part of Figure 63, Notation of declared pins depending on their properties. Req - required, S - single, M - multiple, Token - token multiplicity, Data - data multiplicity. contains required declared data pins, while lower part contains provided data pins of an application. They are distinguished by placement of black rectangle – required data pin has it on the left side, provided data pin has it on the right side. Data multiplicity is depicted by number of triangles – single has one, multiple has three. Token multiplicity is depicted by the colour of the triangles – single has white, multiple has dark. The name of the declared data pin is depicted above the rectangle.

5.2.4 `UnitCall` class

Brief overview. Unit call is used to invoke of a computation unit – start a particular computation step within the whole computation. Unit calls have data pins which denote data sets received as input and output of the computation. Unit calls are part of a CAL program.

Attributes.

Name: string – name of the unit call.

Strength: UnitStrength –unit calls might be Weak=0 or Strong=1. Strong unit calls require all underlying computations to be computed within single *pod*. Weak unit calls do not set any restrictions on computations.

Associations.

Unit [1] – computation unit which is invoked by the unit call.

Pins [*] – computed data pins owned by the unit call. They are copies of the data pins declared by the invoked unit that can be slightly configured according to the needs of current computation.

ParameterValues [*] – values of the unit parameters declared by the invoked computation unit.

Concrete Syntax. Unit calls are depicted as rounded rectangles. Figure 64, Notation of unit calls. Strong unit call on the left, weak unit call on the rightcontains two unit calls. Upper part of the unit call contains unit call's name, the lower part the name of the invoked computation unit release which consists of unit's name and release version. Data pins owned by the unit call are placed on the left and right of the unit call. Required on the left, provided on the right. The strength of the unit call is denoted by outline shape of the rectangle – strong unit calls have solid line, weak – dashed.

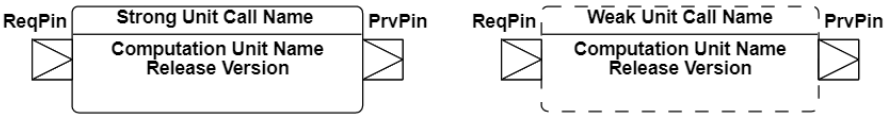

Figure 64, Notation of unit calls. Strong unit call on the left, weak unit call on the right

5.2.5 `ComputedDataPin` class

Brief overview. Computed data pins are not independent CAL elements. Although, computed data pins are part of the CAL program. As the name suggests, the computed data pins are derived from the declaration of the computation unit release a particular unit call is invoking. When unit call is placed in the CAL program, the unit call has computed pins with the same properties as invoked computation release's declared pins. Basically, computed pins represent declared pins in the context of the unit call. Computed data pins are used as places where incoming and outgoing data flows are connected to the unit call. There are just few possibilities to configure computed data pins – one can name pins more appropriate for the computation and set the pin group and depth for more complicated parallel processing cases.

Generalizations. It is specialized from `DataPin` class.

Attributes.

[*inh*] Name: string – specifies the name of the data pin.

[*inh*] Uid: string – specifies the GUID of the data pin.

[*inh*] TokenNo: long – specifies a token number.

[*inh*] Binding: DataBinding – specifies the data binding for the data pin. Binding might be RequiredStrong=0, RequiredWeak=1, Provided=2, ProvidedExternal. Required data pins defines datasets that are consumed, provided – produced. Strong data pins define mandatory datasets, weak – optional.

[*inh*] Type: DataType – specifies type of the dataset, e.g., JSON, XML, Image, etc.

[*inh*] Structure: DataStructure – specifies the data structure, in fact, data schema of the dataset, if the data type is structured.

[*inh*] DataMultiplicity: CMultiplicity – specifies the multiplicity of data items in the dataset. It might be Single=0, Multiple=1.

[*inh*] TokenMultiplicity: CMultiplicity – specifies the multiplicity of tokens consumed or produced by the data pin. It might be Single=0, Multiple=1.

[*inh*] Access: AccessType – specifies way to access dataset, e.g., MongoDB, FTP, etc. …

Associations.

[*inh*] Incoming [0..1] – incoming data flow. Provided computed data pins (Binding = RequiredWeak or RequiredStrong) may not have incoming data flows.

[*inh*] Outgoing [0..1] – outgoing data flow. Required computed data pins (Binding = Provided or ProvidedExternal) may not have incoming data flows.

Call [1] – unit call which owns the computed pin.

Group [0..1] – pin group the computed pin belongs to.

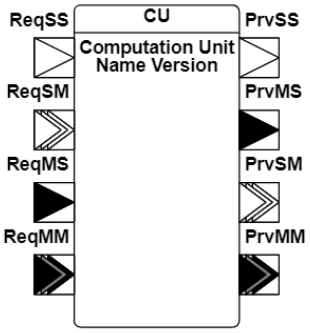

Figure 65, Notation of computed data pins attached to the call unit. Req – required, Prv – provided, two last letters provide the data and token multiplicity accordingly. S – single, M – multiple.

Concrete Syntax. Computed pins are depicted as rectangles pinned to the unit call box (See Figure 65, Notation of computed data pins attached to the call unit. Req – required, Prv – provided, two last letters provide the data and token multiplicity accordingly. S – single, M – multiple.). There is no distinction between required and provided data pins except the placement regarding the unit call's box – required are on the left, provided on the right. Notion of other properties of computed pins is identical to the notion of declared data pins. Data multiplicity is depicted by number of triangles – single has one, multiple has three. Token multiplicity is depicted by the colour of the triangles – single has white, multiple has dark. The name of the declared data pin is depicted above the rectangle.

5.2.6 `DataPin` class

Brief overview. Data pin is an abstraction of declared and computed data pins. It holds most of attributes and associations of both types of pins. Data pins may be connected by the dataflows. Every data pin might have just one data flow connected to it. Declared pins represents input and output of the CAL program (which is computation application as well). Required declared pins start the data flow and provided declared data pins end. Computed pins do it as well, but they act in an opposite way – required computed pins end the part of dataflow delivering data tokens to the computation unit, but provided computed pins starts the part of dataflow receiving computed data tokens from the computation unit.

Generalizations. It has specializations to `DeclaredDataPin and ComputedDataPin` classes.

Attributes.

Name: string – specifies the name of the data pin.

Uid: string – specifies the GUID of the data pin.

TokenNo: long – specifies a token number.

Binding: DataBinding – specifies the data binding for the data pin. Binding might be RequiredStrong=0, RequiredWeak=1, Provided=2, ProvidedExternal. Required data pins defines datasets that are consumed, provided – produced. Strong data pins define mandatory datasets, weak – optional.

Type: DataType – specifies type of the dataset, e.g., JSON, XML, Image, etc.

Structure: DataStructure – specifies the data structure, in fact, data schema of the dataset, if the data type is structured.

DataMultiplicity: CMultiplicity – specifies the multiplicity of data items in the dataset. It might be Single=0, Multiple=1.

TokenMultiplicity: CMultiplicity – specifies the multiplicity of tokens consumed or produced by the data pin. It might be Single=0, Multiple=1.

Access: AccessType – specifies way to access dataset, e.g., MongoDB, FTP, etc. …

Associations.

Incoming [0..1] – incoming data flow. Required declared data pins and provided computed data pins may not have incoming data flows.

Outgoing [0..1] – outgoing data flow. Provided declared data pins and required computed data pins may not have outgoing data flows.

5.2.7 `DataFlow` class

Brief overview. Data flows are used to depict the transition of data tokens between data pins. Data flow connects exactly two data pins.

Associations.

Target [1] – data pin which consumes data tokens.

Source [1] – data pin which produces data tokens.

Constraints. Data flow may connect data pins with compatible data types and the same access type. Depending on data pin class (Decl. – declared, Comp. – computed) and binding (Req. – required, Prv. – provided) the data flow is allowed accordingly to this table:

To / From	Decl. Req.	Decl. Prv.	Comp. Req.	Comp. Prv.
Decl. Req.	No	Yes	Yes	No
Decl. Prv.	No	No	No	No
Comp. Req.	No	No	No	No
Comp. Prv.	No	Yes	Yes	No

Concrete Syntax. Data flows are depicted as arrows. They must connect two data pins. (See Figure 66, Simple example of CAL program - two declared pins, a unit call with two computed pins, and two data flows.

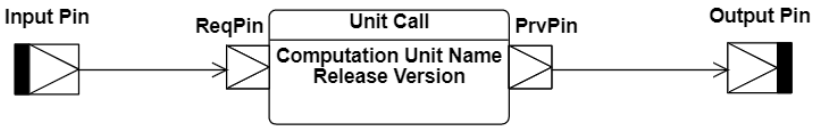

Figure 66, Simple example of CAL program - two declared pins, a unit call with two computed pins, and two data flows.

5.3 CAL Types

There are several enumerations and classes used by CAL elements to describe the particular property. Enumerations (`UnitReleaseStatus`, `UnitStrength`, `DataBinding`, `CMultiplicity`) has been introduced in the previous sections and they have fixed lists of possible values. However, there are several classes which describes undisclosed lists with possible values for CAL element properties. They are `DataType`, `AccessType`, `DataStructure`.

5.3.1 `CalType` class

Brief overview. CAL type is an abstraction of the values of data types, access types and data structures.

Generalizations. It has specializations to `DataType`, `AccessType` and `DataStructure` classes.

Attributes.

Name: string – specifies the name of the *CAL type*.

Uid: string – specifies the GUID of the *CAL type*.

Version: string – specifies a version of the *CAL type*.

IsBuiltIn: bool – specifies if the *CAL type* has a built-in support in the BalticLSC Environment. It allows copying data to the cluster where computation executes, and a module can access it within the same *pod*.

5.3.2 `DataType` class

Brief overview. Data type is allowed value for the dataset and data pin property `Type`. It describes a particular type of data, and its version. E.g., JSON, XML, Image. Data types are used as constraints on data flows allowing transitions from and to pins of the compatible data type. Data types are also used to match data pins with datasets when starting the computation tasks.

Generalizations. It is specialized from `CalType` class.

Attributes.

[*inh*] Name: string – specifies the name of the data type.

[*inh*] Uid: string – specifies the GUID of the data type.

[*inh*] Version: string – specifies a version of the data type.

[*inh*] IsBuiltIn: bool – specifies if the data type has a built-in support in the BalticLSC Environment. It allows copying data to the cluster where computation executes, and a module can access it within the same *pod*.

isStructured: bool – specifies whether this type is structured. E.g., for XML data the data schema might be denoted for data pin or dataset.

CompatibleAccessTypeUids: List<string> - list of compatible access types.

ParentUid: string – specifies the parent data type of the given type. Thus, data types make hierarchy, and it is possible to specify more general data type, e.g., Image, rather than more specific JPG.

5.3.3 `DataStructure` class

Brief overview. Data structure is allowed value for the dataset and data pin property `Structure`. It describes a particular structure of data, and its version. Data structures, basically, are data schemas that defines the structure of data using the means available for the data type, e.g., JSON Schema may be used for JSON data.

Generalizations. It is specialized from `CalType` class.

Attributes.

[*inh*] Name: string – specifies the name of the data structure.

[*inh*] Uid: string – specifies the GUID of the data structure.

[*inh*] Version: string – specifies a version of the data structure.

[*inh*] IsBuiltIn: bool – specifies if the data structure has a built-in support in the BalticLSC Environment.

DataSchema: string – specifies the data schema of the data structure. Schema itself might be coded accordingly to the data type it is applied, e.g., JSON Schema may be used for JSON data.

5.3.4 `AccessType` class

Brief overview. Access type is allowed value for the dataset and data pin property `Access`. It describes a type of access to the data, and its version. Type of access is a technology or protocol used to access data, e.g., FTP protocol might be used to access images.

Generalizations. It is specialized from `CalType` class.

Attributes.

[*inh*] Name: string – specifies the name of the access type.

[*inh*] Uid: string – specifies the GUID of the access type.

[*inh*] Version: string – specifies a version of the access type.

[*inh*] IsBuiltIn: bool – specifies if the access type has a built-in support in the BalticLSC Environment. It allows copying data to the cluster where computation executes, and a module can access it within the same *pod*.

AccessSchema: string – specifies the access schema of the access type. It is a JSON Schema that describes what access information is needed to access type, e.g., host, username and password are needed to access FTP server.

PathSchema: string – specifies where within the given resource is the dataset located. E.g., folder for FTP server or collection for MongoDB database.

StorageUid: string – specifies GUID of the storage.

ParentUid: string – specifies the parent access type of the given type. Thus, access types make hierarchy, and it is possible to specify more general access type, e.g., NoSQL database, rather than more specific MongoDB.

5.4 CAL Semantics and Examples

In this section we present the semantics of CAL through describing two basic mechanisms.

1. Functioning of Computation Module instances that results from the execution of Unit Calls.
2. Functioning of the Computation Engine that manages execution of the Computation Application Release program graph.

This semantics is illustrated by using example Computation Module definitions and simple CAL applications.

5.4.1 Execution of Computation Modules through Unit Calls

A ***Computation Module*** is a self-contained piece of software that performs a well-defined computation algorithm. It operates on ***Data Sets***, like files, folders, database collections, database tables etc. Access to these Data Sets is communicated using ***Data Tokens*** passed through ***Data Pins***. Data Pins of type "input" accept Data Tokens that point to Data Sets provided by other modules or directly by the user. Data Pins of type "output" deliver Data Tokens that point to Data Sets provided by the current module.

Computation Modules should have many (at least one) input pins and can have many (maybe none) output pins. The case when there are no output pins is reserved for special modules that can communicate directly with data stores outside of the BalticLSC system. Execution of a Computation Module within a CAL program is denoted by a Unit Call associated with the module. The module starts its operations (a module instance is created) when it receives a Data Token on every of its input pins. When the module finishes its computations (all or part), it sends a Data Token on a specific output pin.

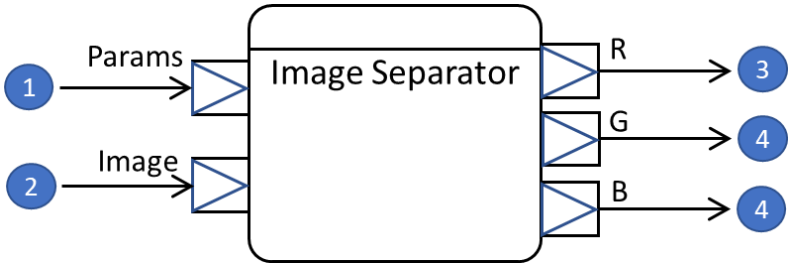

Figure 67 Example Computation Module with "single" Data Pins

This behavior is illustrated in Figure 67. The module ("Image Separator") receives two tokens (denoted as "1" and "2") on its input pins (named "Params" and "Image"). When it finishes processing

(executing a computation algorithm), it sends three tokens (denoted as "3", "4" and "5") to its output pins (named "R", "G" and "B").

The module in Figure 67 contains only "single" Data Pins. Such pins assume that only one token is passed through them for each execution of a computation algorithm. For instance, the example "Image Separator" accepts one "Image" token with an associated "Params" token. Then it processes this single image and produces one resulting token for each of the three image components ("R", "G" and "B").

Data Pins can be also defined as "multiple". This can pertain to "data multiplicity" or "token multiplicity". Pins with multiple data, accept or produce tokens that refer to data collections (e.g., folders vs. single files). Pins with multiple tokens accept or produce sequences of tokens within a single computation execution. A computation module with pins of the "multiple" kind is illustrated in Figure 68. The "Image Tokenizer" accepts a single token (denoted as "1") that refers to a collection (a folder with image files, cf. data multiplicity). Then, it produces multiple tokens (denoted as "2-1" and "2-2", cf. token multiplicity) that refer to individual image files. Note that pins can be "multiple" both for data and for tokens.

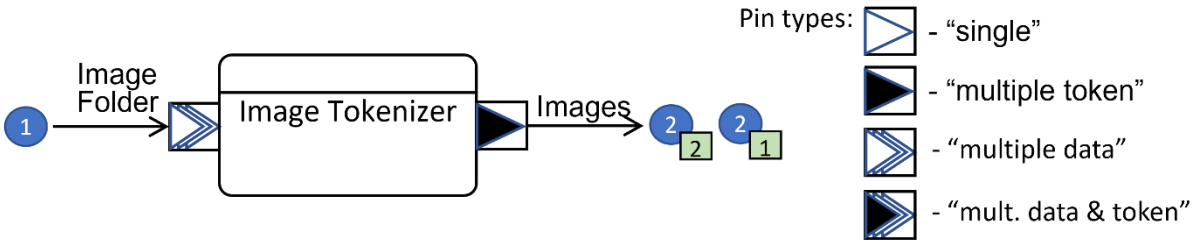

Figure 68 Example Computation Module with "multiple" Data Pins

Considering configuration and multiplicity of input and output pins, we can distinguish several reference module types. These types can be combined into hybrid types (e.g., splitter-joiner).

- Simple processor (one single input, one single output)
- Data separator (one single input, many single outputs)
- Data splitter (one single input, at least one token-multiple output)
- Data joiner (many single inputs, one single output)
- Data merger (at least one token-multiple input, one single output)

5.4.2 Management of Computation Application Release execution

Execution of Computation Modules within a Computation Application Release is managed by a Computation Engine that operates according to the rules described below. We will illustrate these rules with simple examples.

The first example is a very simple program, shown in Figure 69. The program contains a single Unit Call, which, according to the language syntax, contains two compartments. The first compartment contains the call name (here: "detector", it can be any name that identifies this particular call). The second compartment contains the name of a particular Computation Module, selected from the list of available modules, that is called by this call (here: Face Detector).

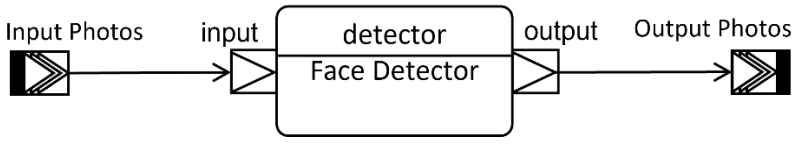

Figure 69 Simple CAL application with one Module Call and two External Data Pins

The Module Call in Figure 69 contains two "single" Data Pins ("input" and "output"), where their meaning was explained in the previous chapter. Apart from this, the program contains two Declared Data Pins ("Input Photos" and "Output Photos"). These pins refer to some data sources external to the execution environment. When the application is executed, the user can select specific credentials and location (e.g., URL) where these data sources are situated.

Pins are connected through Data Flows denoted with arrows. Note, that in our first example, a "multiple data" pin ("Input Photos") is connected to a "single" pin ("input"). This means that the execution engine will process several data items (e.g., files) stored in a collection (e.g., a folder). For each such data item, a separate instance of the Face Detection module will be started. Each such instance will receive a token with a reference to a single data item (file) in the collection (folder). This means that the engine will possibly execute several instances of the Face Detector module in parallel, each processing a single data item (here: a photo file).

When any of the Face Detector module instances finishes processing its input data item, it sends a token to its "output" pin. This token will then be passed to the "Output Photos" pin. Note that the "Output Photos" is a "multiple data" pin, just like the "Input Photos" pin. This means that the data item will be sent to a collection (folder). In summary, the program in Figure 69 takes a single folder placed on some server managed by the user. It then sends photo files from that folder to several instances of the Face Detector module that run in parallel. While each of the module instances finish, the results of their work are placed in some (perhaps other) folder provided by the user.

Another, more complex CAL program is shown in Figure 70. The files in the "Input Images" folder

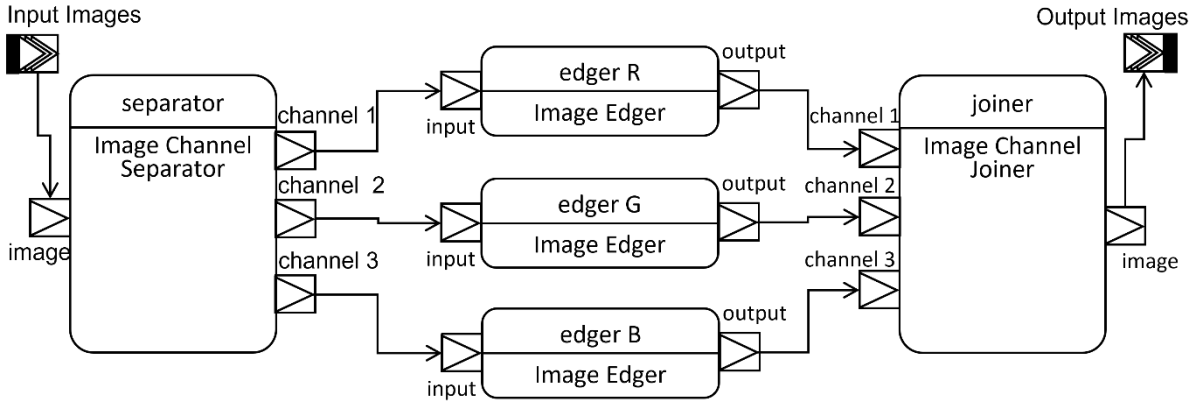

Figure 70 More complex CAL application with several Module Calls

are sent to several "Image Channel Separator" module instances that run in parallel. Each such instance takes an "image" on its input and produces three images, sending three tokens – one on each of its output pins ("channel 1-3"). Each of these three images are in turn processed by a separate instance of the "Image Edger" module. When three "edger" instances finish their work, their "output" tokens are passed to an instance of the "Image Channel Joiner" module. This module takes the three "channel" images (produced by the "edgers"), combines them into a single image and sends a token on its output (the "image" pin). All the images produced by the instances of the "Image channel Joiner" module are then stored in the folder determined by the "Output Image" pin.

It should be noted that the program in Figure 70 can cause execution of many module instances in parallel. For example, let's consider running the application with 10 images stored in the "Input Images" folder. This will result in parallel execution of up to 10 instances of "Image Channel Separator", 30 instances of "Image Edger" and 10 instances of "Image Channel Joiner". Depending on the computation complexity and processing time for individual photos, it might happen that some instances of "Image channel Joiner" might start (and even finish) their tasks, while some of the instances of "Image Channel Separator" will still work. Also, it should be noted that the CAL execution engine will properly handle all the tokens flowing within the system. This means for instance, that the "channel" tokens for a single image will be separated and then joined appropriately, preventing from "mixing" tokens that "belong" to the same initial image.

6. Conclusion

The main objective of the document is to describe the design for BalticLSC Software. The design specification is a “live” document, and it is updated during the process of software design and development. The main work is the addition of the necessary details to the behavioural and data models according to the work plan of the agile design and development process.

This document is the main source of the information for the ongoing activities A6.2 Implementation of the BalticLSC Software to ensure the match between design and developed software.