



EESSI – RINA

Business Use Case Engine Architecture

Architecture & Design Specifications

Table of Contents

1	Introduction.....	6
1.1	Scope.....	6
1.2	Structure of the document	6
1.3	Reference and Applicable documents.....	6
1.4	Audience	7
1.5	Definitions, Acronyms and Abbreviations	7
2	Architecture Overview.....	9
2.1	Architecture Diagram	9
2.2	BUC-Engine Components	10
3	Functionality	11
3.1	Actions.....	11
3.1.1	Action Execution.....	12
3.1.2	ActionHandlers	13
3.1.3	EventListeners	14
3.1.4	Triggers and TriggerHandlers	14
3.2	MessageDispatcher	14
3.3	Persistence	14
3.4	Concurrency.....	15
3.5	Version Loading.....	16
3.6	Plugins	17
3.6.1	DocumentSenderPlugin	18
3.7	Document Content.....	18
4	Integration	18
4.1	Engine Initialization	18
4.1.1	Configuration File	19
5	BUC Flow testing	20
5.1	JUnit Runner	20
5.2	Flow Test Definition	20
5.2.1	Document Content Mocking	21
5.2.2	Payload Provider.....	21
5.3	Flow Test Integration	21
6	Service Contract.....	23
7	Data Contract	24
8	Exception Managements.....	25
8.1	ActionExecutionException	25
8.2	BusinessValidationException.....	25
9	Logging Management.....	26
9.1	Logger types	26
9.1.1	BUC-Engine Loggers	26
9.1.2	BUC-Processes Loggers	26

Table of Figures

Figure 1: RINA BUC Engine Architecture	9
Figure 2: RINA BUC Engine Components	11
Figure 3: RINA BUC Engine Action's Flow	12
Figure 4: BUC Engine Locking Approach	16
Figure 5: RINA BUC Process Artifacts Class Loading	17

Table of Tables

Table 1: References	6
Table 2: Definitions and Acronyms list.....	8

Document Control Information

Document Control	Value
Project Title	Electronic Exchange of Social Security Information (EESSI)
Document Title	EESSI – RINA - Business Use Case Engine Architecture
Document Category	Architecture & Design Specifications
Revision	-
Component Version	-
Last Publication Date Project Milestone	18/12/2020 EESSI-2020
Document Status	Final
Sensitivity (TLP) Distribution terms	Traffic Light Protocol (TLP) = "GREEN"  The distribution of this document is done strictly in line with the Traffic Light Protocol (TLP) established by the European Commission's note AC 790/15 REV for the EESSI project documentation. In line with the note AC 790/15 REV, this document is labelled as TLP = "Green". Therefore, it can be circulated widely within the EESSI community. However, the document or the information herein may not be published or posted on the Internet, nor released outside of the EESSI community.
Connected/Embedded Files	None
Authors	European Commission, DG EMPL F5, EESSI ARCH/RINA
Revised by	European Commission, DG EMPL F5, EESSI QA/QC
Approved by	European Commission, DG EMPL F5, EESSI PMO team

Document history

Project Milestone	Changes/Corrections
	Description
EESSI-2020	Initial Document

1 Introduction

Business use case engine (BUC-Engine) is an in-house developed processing engine for the EESSI business flows (BUCs). From the architecture point of view it consists of the engine itself and the implementation of the BUCs with the custom notation language and processing rules. It replaces the former Bonita BPM software, that has been used as a third-party tool.

1.1 Scope

This document describes the architecture of the BUC-Engine and its components and briefly depicts some of the core components of the BUC-Processes. The later is, however, described in more detail in a separate document "*EESSI 2020 – RINA – BUC Technical Specifications*" [R3].

1.2 Structure of the document

The document is organized as follows:

- Chapter 1: Lists the scope of the document, underlined principles for the specifications, related documents, the audience, definitions, and glossary and how to navigate through the document.
- Chapter 2: Describes the high-level architecture overview.
- Chapter 3: Describes in detail the functionality of the BUC-Engine and it's core components.
- Chapter 4: Provides information about the integration mechanism.
- Chapter 5: Describes the BUC-Engine flow testing framework.
- Chapter 6: Provides basic information about the service contract
- Chapter 7: Provides basic information about the data contract
- Chapter 8: Describes the exception management and handling.
- Chapter 9: Describes the logging and auditing.

1.3 Reference and Applicable documents

Ref Id	Reference
[R1]	EESSI Glossary
[R2]	EESSI 2020 – RINA – Architecture Overview
[R3]	EESSI 2020 – RINA – BUC Technical Specifications

Table 1: References

1.4 Audience

- RINA Software Architects
- RINA Developers
- Writers of BUC XML Files
- Developers of new BUCs

1.5 Definitions, Acronyms and Abbreviations

Terms and Acronyms	Definitions
BUC	Business Use Case: is a single set of rules and steps describing a process as defined in the BUC-Guidelines, which originate from the EU regulation EC 883/2004
SED	Structured Electronic Document - is a unit being exchanged within a BUC. Can be also used only as Document.
BUC-Model	Defines the rules of the modelling of a BUC as XSD schemas
BUC-Engine	Implementation of the common module, which runs the BUC implementations.
BUC-Definition	A formal definition of the BUC processing rules (as an XML, or a BPMN diagram).
BUC-Implementation	An implementation of a particular BUC for the BUC-Engine to run, which encapsulates the BUC-Definition files together with custom implementation.
Extension Point	Is an abstract definition of specific BUC processing parts, which can be overridden to provide custom implementation.
Action	in the context of a BUC-Engine represents a single executable unit within the BUC-Definition. E.g. SET-PARTICIPATNS. An Action may be defined for a CASE context of a DOCUMENT context. In the later one, it SHOULD be accompanied by the document type.
ActionHandler	The implementation of the business logic of an Action. It follows a common interface and is registered to handle a particular set of Actions
Event	A notification, which is triggered on specific places during the Action processing.

EventListener	A listener, which defines an implementation, that is executed, when a particular Event is triggered. It can be registered to listen for a specific Event.
Trigger	Every action may trigger various operations like creating a new action or removing an existing action. This triggers are specified within the BUC model.
TriggerHandler	The implementation of a business logic to handle a trigger.
Extension Point ID	To be able to extend even a very specific functionality we need to specify an ID for the extension points. These IDs must refer to the particular context: 1. ActionHandler - ID: BUC type, Action Type, Document Type (optional) 2. Event - ID: BUC type, Action Type, location (BEFORE or AFTER action execution), action result type (SUCCESS or FAILURE) 3. ActionHandler - ID: BUC type, Action Type, Document Type (optional)
Plugin	Is an implementation of a plugin interface, that is used as a business operation within the engine, but the implementation is provided by the user of the BUC-Engine (e.g. DocumentSender)

Table 2: Definitions and Acronyms list

2 Architecture Overview

The new Business Use Case engine (BUC-Engine) together with the implementation of the Business Use Cases (BUCs) provides stateful processing of the BUCs in the EESSI context. The BUC-Engine is part of the RINA fullstack deployment being the core of the Case Processing Services (CPS) layer.

2.1 Architecture Diagram

Following depicts a simplified high-level RINA components diagram:

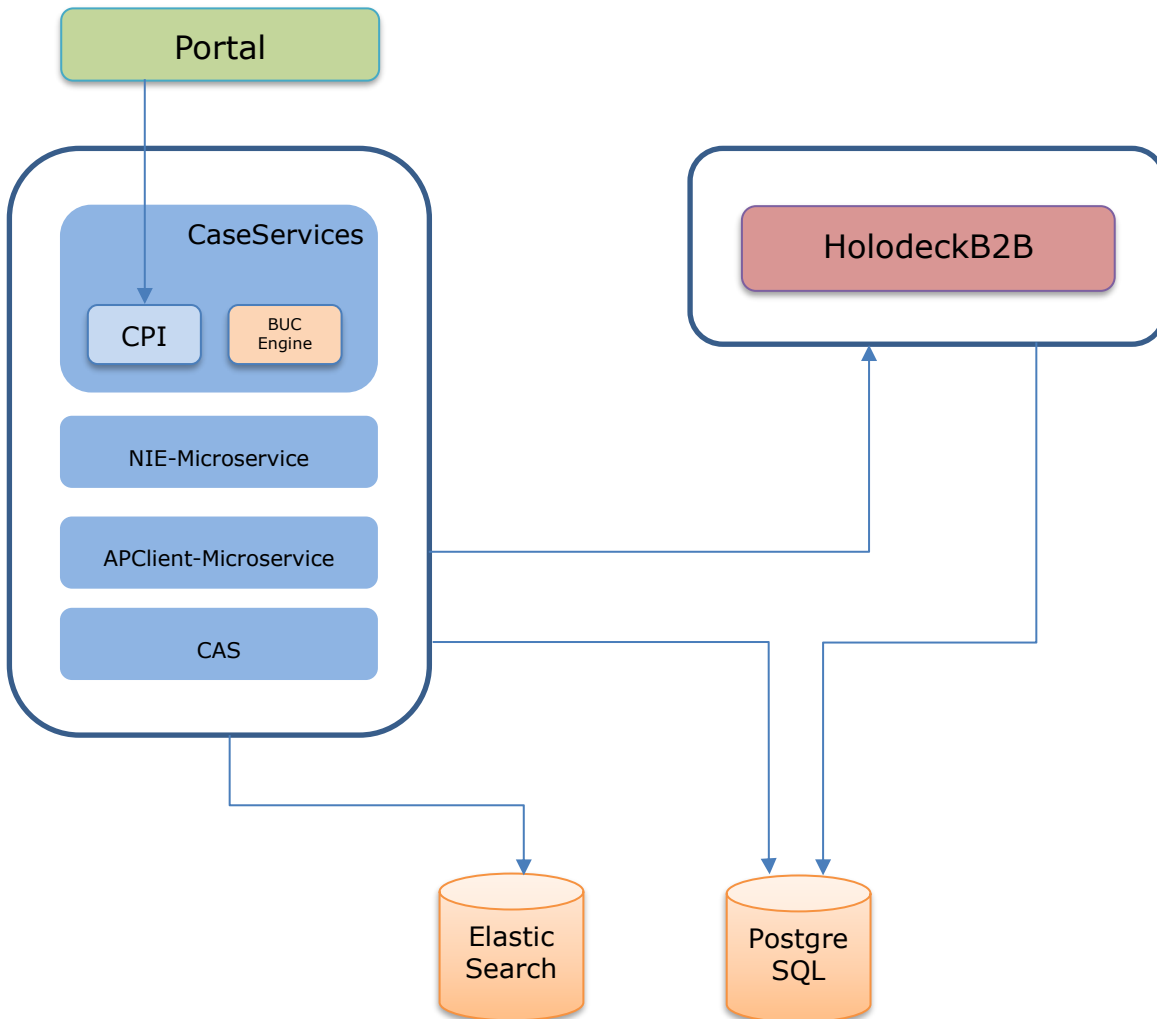


Figure 1: RINA BUC Engine Architecture

The BUC-Engine component resides inside the CaseServices module (Figure 1). The CaseServices module encapsulates the services implementation for processing of cases. In the RINA environment a user (clerk) never communicates directly with the BUC-Engine, rather the BUC-Engine actions are executed through the operations on case services. These are published through the CPI and other public interfaces like NIE.

2.2 BUC-Engine Components

The BUC-Engine component comes with a number of modules, which can be divided into following types:

1. **API** – provides a set of interfaces and common functionality, that can be accessed in all other modules. Defines the structure of the components, which should be implemented by the integration points.
 1. Data Model – set of data objects, that BUC-Engine operates on
 2. BUC Definition Language – set of XSD documents specifying the BUC definition language.
 3. DataSource – set of data source interfaces for various types together with the provider interface which serves as a factory for creating the instances.
 4. Plugins – interfaces of various functional components, that are dependent on the integration and their implementation should be provided by the integration point.
 5. BUC-Engine Functional Components – interfaces for common functional components, which implementation of is provided by the BUC implementations. The BUC-Engine operates on these.
2. **Core** – provides the implementation of the core functionality of the BUC-Engine without any specific context of a particular BUC.
3. **BUC(s)** – provide implementation of the particular BUC(s) following the BUC-Engine API. May introduce custom functionality, which is local to the particular BUC. They are considered a separate artefact, which is “deployed” inside the BUC-Engine.
4. **Flow Tests** – is a separate project, which is built upon the BUC-Engine API and Log4j2 runner. Provides a custom framework for writing and executing BUC Flow Tests – definition of a sequence of steps (actions, assertions, etc.) to be performed within a particular BUC.
5. **In-Memory Datasource** – sample implementation of the Datasources, which provide in-memory persistence. Used mostly in tests.

The following diagram depicts a high-level components architecture (omitting the flow test framework, which is not relevant for this diagram):

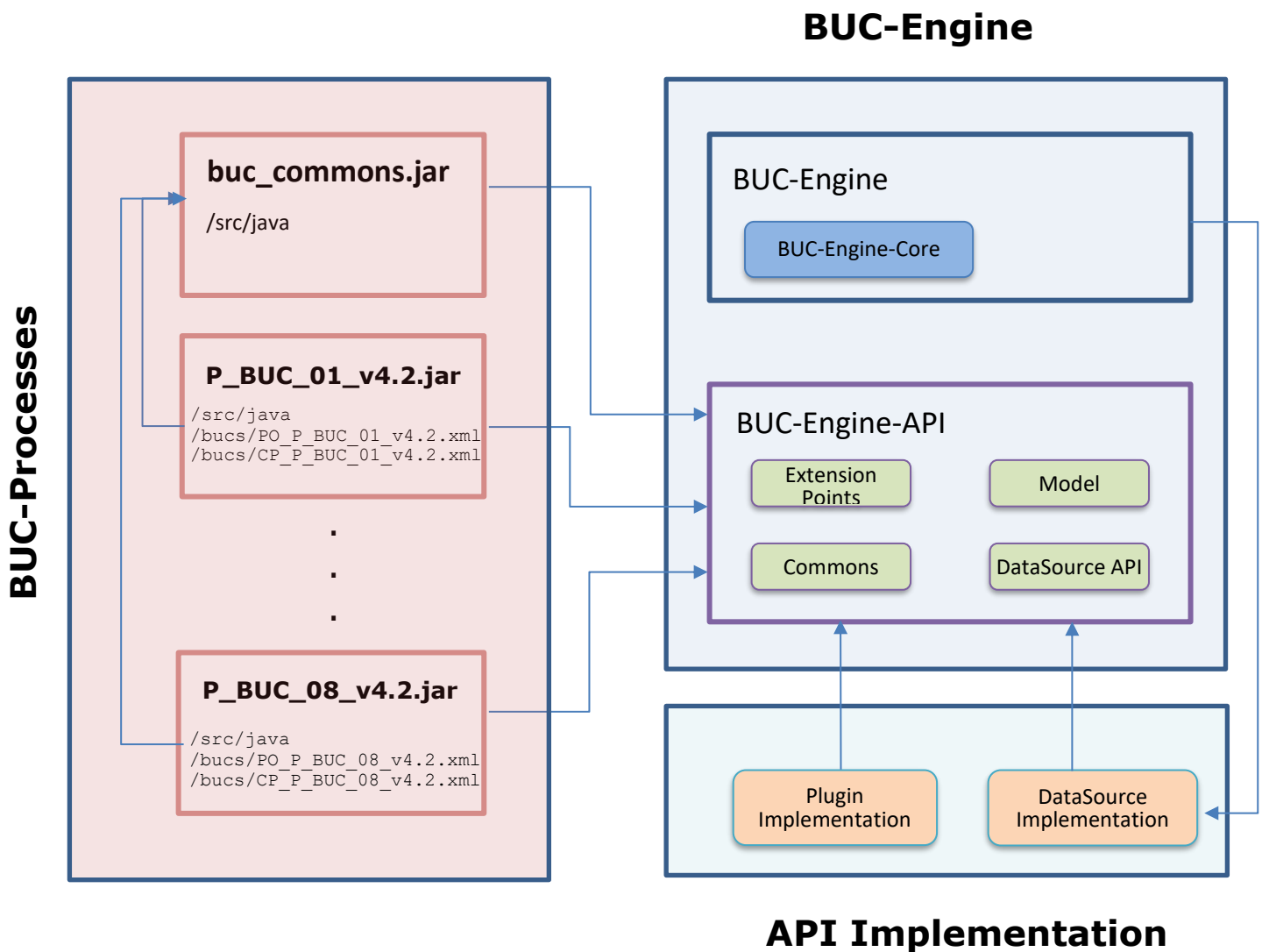


Figure 2: RINA BUC Engine Components

3 Functionality

The BUC-Engine has been designed based on documents and actions, which represent a single unit of execution. Similar to the previous implementation with the Bonita BPM engine, actions can be seen as tasks.

Every BUC definition specifies a set of documents, that share the same processing model, which can be extended. This processing model is based on actions, that might be document or case related.

3.1 Actions

Actions are encoded in:

1. BUC Process definition file (i.e. an XML file)
2. In the Java code (either common or specific to a single BUC)

The interaction with the BUC-Engine is always performed through execution of these actions, where every action is identified by an unique ID and may require a specific payload type, which should be provided. The high level processing of an action is defined globally within the `buc-engine-core` module and is the same for every action. However, the specific behavior of an action is provided in it's designated `ActionHandler`.

3.1.1 Action Execution

Following is a diagram describing the common processing model of an action:

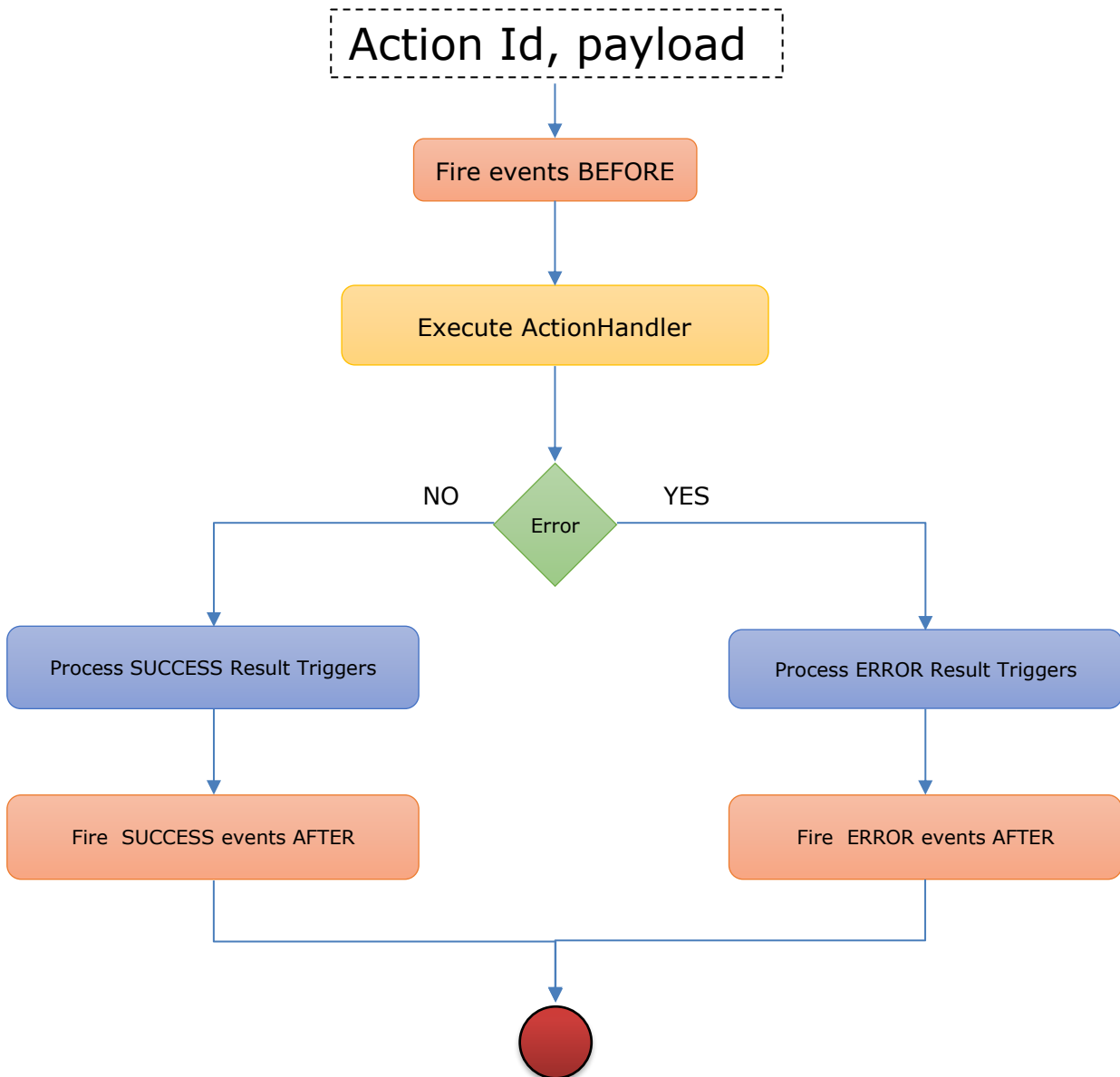


Figure 3: RINA BUC Engine Action's Flow

As mentioned above, every interaction with the process within the BUC-Engine happens through an execution of an action. There are, however, three different use cases, that are slightly different in their nature:

1. **Create a case** – because the case does not exist yet, there is also no action to be executed. Hence an action object with the specific type `CASE_CREATE` is constructed and executed. The implementation of the handler is provided in the BUC-Processes as usual.
2. **Execute action** – the engine is given an action identifier and the payload and it will create the action object based on the metadata of the existing action
3. **Dispatch a message** – when a business message is received, the correct action to be executed must be identified based on the attributes it is carrying.

These use cases differ mostly in the way how the correct action is constructed. All the use cases at the end call the same method to execute the action and the business logic of each action is provided within the BUC-Processes implementation. This way the high level separation of concerns is achieved.

An EESSI BUC flow defines a set of documents, which can be sent and received. Although, the processing model for every document is common, there are a lot of exceptions and custom flows. To allow for this kind of versatility, there are several extension points, which might be defined for every action:

1. `ActionHandler`
2. `EventListener`
3. `TriggerHandler` and `Trigger` definition

It is important to note, that the BUC-Engine core module is agnostic of any implementation of the process, hence is kept very simple and lightweight. The specific extension points are registered in so-called registries in a hierarchy to allow for overriding of less specific handlers. For a detailed description of the modelling guidelines please refer to "*EESSI 2020 – RINA – BUC Technical Specifications*" [R3].

3.1.2 ActionHandlers

For every action, there must be an `ActionHandler` defined, which contains the business logic performed when the action is executed.

The `ActionHandlers` are specific to the BUC-Processes and they practically define the business flow of a BUC (together with other functional components). Therefore, the implementation of the `ActionHandlers` resides in the BUC-Processes implementation and is deployed as part of the BUC-Process.

This boundary between the business logic of an action and the BUC-Engine execution allows to create `ActionHandlers`, which are specific to a single action, BUC definition or even a particular model version. In other words, calling a `SEND` action on a document for one BUC might be implemented differently between the BUCs. Such model allows for great versatility of the behaviour, if needed.

All `ActionHandlers` implement a common interface defined in the `buc-engine-api` module.

3.1.3 EventListeners

As depicted in the action execution diagram, there are three types of events fired during an action execution:

1. Before the action is executed
2. After a successful action execution
3. After a failure action execution

An `EventListener` implementation can be registered to handle any of the above-mentioned events for a specific action type of a specific document type for a specific BUC type.

3.1.4 Triggers and TriggerHandlers

Every document from a BUC-Definition might specify a set of triggers. These triggers are fired based on the action performed and provide a functionality to be executed. There is a collection of specific triggers provided in the `buc-engine-api`:

1. `CreateActionTrigger` – specifies an action to be created
2. `RemoveActionTrigger` – specifies an action to be removed
3. `SuspendActionTrigger` – specifies an action to be suspended
4. `ReactivateActionTrigger` – specifies an action to be reactivated

These triggers are processed by `TriggerHandlers`, which follow the same logic as `ActionHandlers` and are part of the BUC-Processes. They also allow to register a trigger handler for a specific BUC, version, action types, etc. The difference between an `ActionHandler` and a `TriggerHandler` is, that a `TriggerHandler` cannot be called from outside. A `TriggerHandler` is executed always as a direct consequence of an action.

3.2 MessageDispatcher

The interaction with the BUC-Engine happens through actions. Usually these actions are triggered by a client, for example an UI that is controlled by a clerk. A clerk can see all the actions for a specific case and might choose one to execute, providing also the payload if needed.

Such an interaction, however, is missing in case of a message reception, where there is no clerk involved. The BUC-Engine was designed in such a way, that also a receiving a message is a specific action, which must exist prior to calling it. This interaction is handled through a `MessageDispatcher` component in the BUC-Engine, which is responsible for executing the correct action based on the reception of a message. If no such action can be found, the message will not be dispatched and it is up to the caller to decide how to handle such case.

3.3 Persistence

The BUC-Engine data model is defined in the `buc-engine-api` module. Both, the BUC-Engine and the BUC-Processes use this model to represent metadata of documents, cases, etc. The persistence mechanism and the underlying data storage is left for the implementation by the integration point. There are no implications on the technologies used.

BUC-Engine accesses the data through a set of well-defined interfaces called `DataSource(s)` and a `DatasourceProvider`. An implementation of these interfaces must be provided by the integration point during the initialization of the engine.

This pattern allows for usage of any persistence mechanism desired, simply by implementing the needed data sources. There is, however, an implication on the transactional behavior of the BUC-Engine. Because the engine is agnostic of any storage mechanism, it also does not use any transaction management and leaves this to the caller of the engine.

As a result, if a transaction management is used, every BUC action execution should be treated as a single transaction. Not doing so might result in an inconsistent state.

3.4 Concurrency

The BUC-Engine, as well as the BUC-Processes have been implemented in a thread-safe manner. The implementation as such allows for concurrent processing of actions, given that the persistence layer is also implemented in such way.

The basic rule for the BUC-Engine concurrency is, that any single action might be executed only once in a particular moment of time.

The BUC-Engine provides a distributed action locking mechanism through a `Hazelcast` map. It does not perform any locking by itself, but exposes methods to lock/unlock an action. It is always up to the integration point to correctly use those in it's context.

The BUC-Engine application interface provides three methods for concurrency locking `lockAction`, `releaseLockAction` and `isActionLocked`:

```
/**
 * Public method to lock the action at the beginning of the execution to
 * prevent other threads for executing the same action during this time.
 *
 * @param actionId
 * @return true/false
 */
boolean lockAction(String actionId);

/**
 * Unlock the action at the end of the execution to allow other threads to
 * execute this action if necessary only if there is no other thread using
 * it at the current moment.
 *
 * @param actionId
 */
void releaseLockedAction(String actionId);

/**
 * Check to see if an action is locked by the current thread prior to
 * starting the execution. This is used inside the execution (engine). If
 * the current thread is not the one who locked the action or the action is
 * not locked at all than a BucEngineException is thrown.
 *
 * @param actionId
 * @return true/false
 */
boolean isActionLocked(String actionId);
```

Next diagram depicts the concurrent action execution:

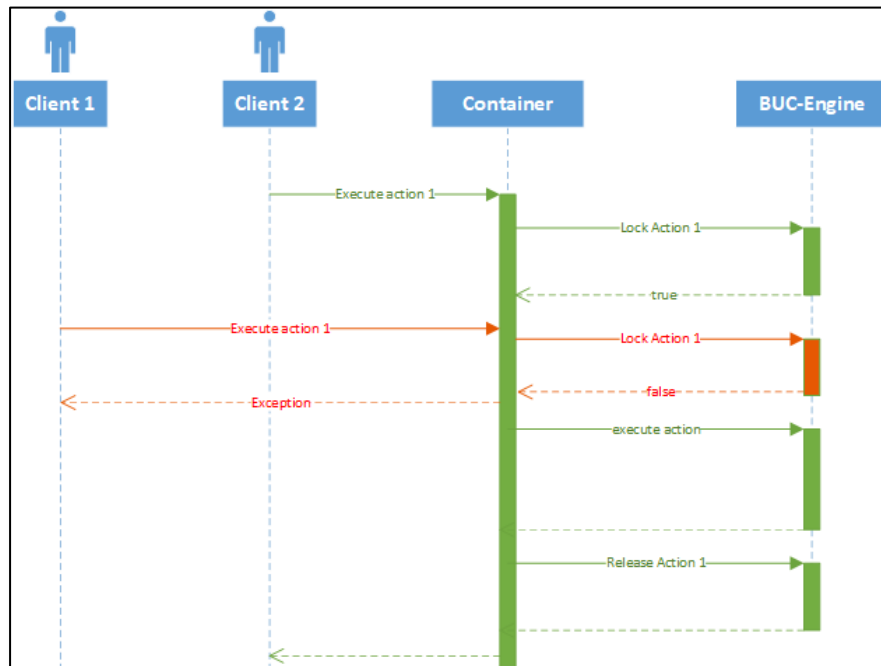


Figure 4: BUC Engine Locking Approach

3.5 Version Loading

The EESSI BUC Processes follow a versioning model, that allows multiple different version of the same processes to be running in parallel. Each version might provide a different implementation even for the same action (e.g. sending a document of type P2000 for the buc P_BUC_01 in the version 4.1 might require to set some document metadata, but in the version 4.2 it does not).

To allow a different behavior for the same process, the BUC-Engine implements a custom Java Classloader, which takes care of loading the correct implementation of the BUC Process artefacts depending on the model version. In fact, there is an instance of a `BucVersionURLClassLoader` created for every model version found in the processes directory. The initialization of the classloader follows a convention, where the versions are located in the configured directory, i.e.:

```

/NAS/processes/
| 4.1
| | p_buc_01_v4.1.jar
| | p_buc_02_v4.1.jar
| 4.2
| | p_buc_01_v4.2.jar
| | p_buc_01_v4.2.jar
  
```

The BUC-Engine will check the subdirectories of the directory configured through the property `buc.archive.folder` and will create a version classloader for every directory that matches the patter `<major>.<minor>`.

An instance of the `BucVersionURLClassLoader` is provided with a directory containing the processes artifacts including all the dependencies besides those, which are already

provided through the BUC-Engine. The parent classloader of each such version classloader instance is the context classloader, hence first loading all the classes provided through the BUC-Engine and only if the desired class is not found, it will refer to the configured directory. This mechanism allows to replace implementation or fix bugs only for a specific model version or even a single BUC Process implementation, without touching the other ones.

The following diagram displays the BUC Process artifacts classloading:

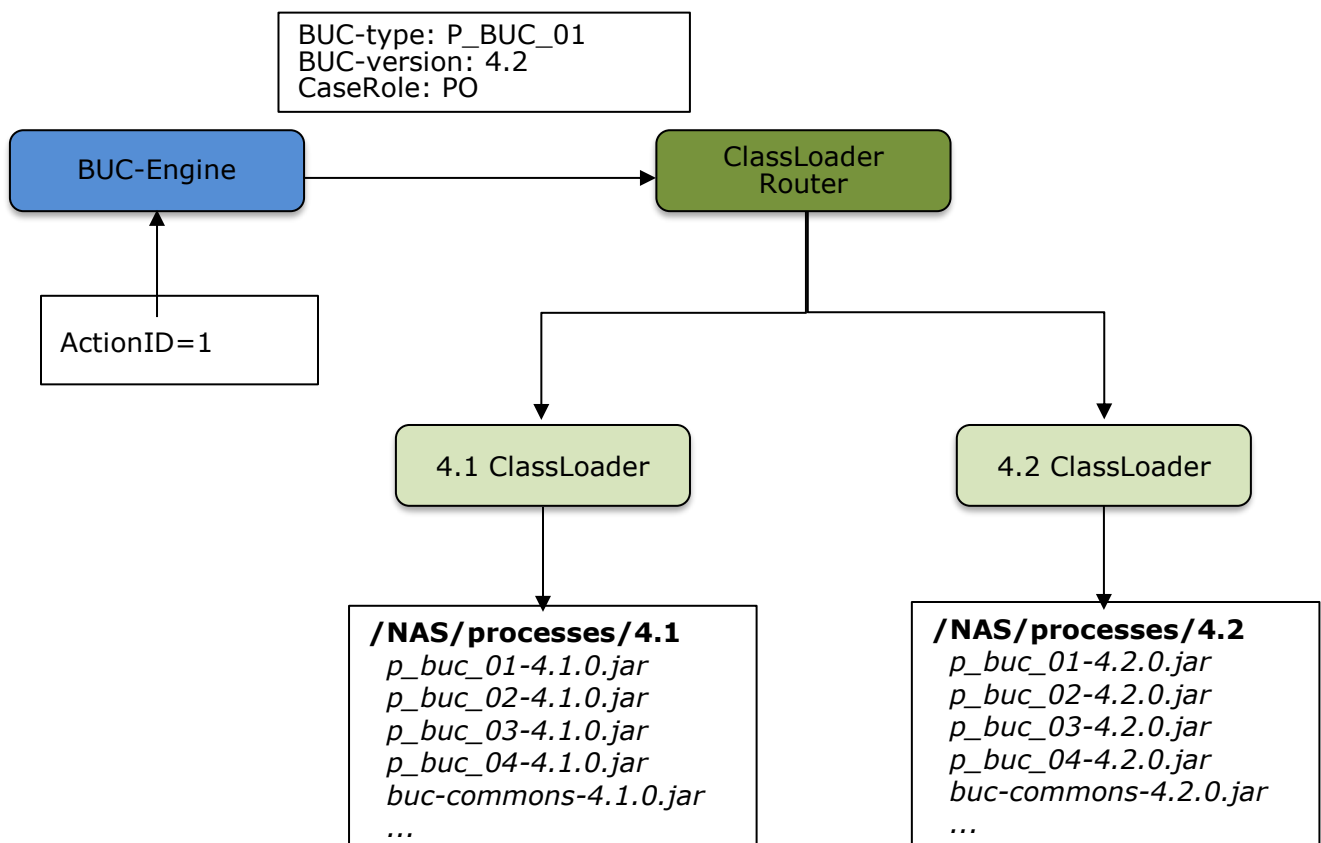


Figure 5: RINA BUC Process Artifacts Class Loading

3.6 Plugins

The BUC-Engine architecture has been designed to deal exclusively with the BUC flow processing of the cases.

There are, however, functionalities which are triggered by the BUC flow, but the implementation is dependent on the integration. The implementation of such functionalities is provided as plugins.

A plugin, in the context of the BUC-Engine, is always defined as an interface, where the implementation is provided by the integration point. Currently, there is only one external implementation of a plugin needed – for sending a message. All plugins must be kept thread-safe as the same instance is used by all the processes.

3.6.1 DocumentSenderPlugin

Sending a document is an operation, which is triggered by the execution of an action and as such the action is implemented in the BUC Processes implementation.

To not impose any explicit restrictions on the sending part, the BUC-Engine calls the implementation of the `DocumentSenderPlugin` when a message is sent. It is the responsibility of the plugin to construct the SBDH and perform any steps needed to successfully send a message. The plugin method returns the full message object, which can be further processed by the BUC-Engine.

There are two ways to register the sender plugin:

1. Automatic loading – is provided by the Java service loading mechanism. To load the plugin implementation automatically it should be placed on the classpath of the BUC-Engine and a services file `eu.ec.dgempl.eessi.rina.buc.api.plugin.DocumentSenderPlugin` should be placed into the META-INF/services directory. The file should contain the full package and class name of the plugin implementation. For further documentation refer to the Java documentation of the `ServiceLoader<S>` class.
2. Manual plugin registration – after creating an instance of the `BucEngine`, a plugin might be registered by calling the method `BucEngine.registerPlugin` with the plugin instance.

3.7 Document Content

The very essence of the EESSI eco-system are the documents (SEDs), which carry the structured content between the case participants.

In most cases, the BUC flow is navigated based on the document types and case metadata. However, for some business decisions a parameter from the document content might be needed.

The BUC-Engine never creates or in any other way mutates the document content. It has, however, a read access to the document content datasource. Therefore, any document content, which might play a role in the BUC-Engine flow should be handled prior to calling the BUC-Engine action.

4 Integration

This section describes the integration of the BUC-Engine in any container. In the RINA context, the BUC-Engine is integrated in the Case Processing Services (CPS). The BUC-Engine gets integrated through an instance of the `BucEngine`. Currently, there is only one implementing class `BucEngineApplication`.

4.1 Engine Initialization

Following is needed to successfully initialize the BUC-Engine:

1. Configuration file
2. Implementation of the `DatasourceProvider`.
3. `DocumentSenderPlugin` implementation

4. BUC-Process Artifacts

4.1.1 Configuration File

The configuration file contains key/value pairs of various BUC-Engine properties. RINA Case Services requires a file path to the BUC-Engine configuration file to be provided as the following JVM argument:

```
-Drina.configuration.bucengine=<path-to-the-file>
```

The file may contain following configuration properties:

1. `buc.archive.folder` – a path to the folder, containing the BUC archives. This folder must contain a directories with the name of every model version that is deployed, where the BUC archives are stored, i.e.:

```
/NAS/processes/  
| 4.1  
|   p_buc_01_v4.1.jar  
|   p_buc_02_v4.1.jar  
| 4.2  
|   p_buc_01_v4.2.jar  
|   p_buc_01_v4.2.jar
```

2. `buc.archive.version` – if the `buc.archive.folder` property is not defined, this property explicitly defines the model version and will only use the context classloader instead of the `BucVersionURLClassLoader`.
3. `sed.xsd.folder` – defines the full path to the directory containing the XSD files for the SEDs.
4. `hazelcast.config.file` – defines a full path to the Hazelcast configuration file. If the path is not provided, a new default Hazelcast instance will be create.

5 BUC Flow testing

In the context of the EESSI, there are currently cca. 100 different BUC process types for a single model version and each one of them defines a flow for the case-owner and counter-party.

With multiple model versions and different scenarios within a single case the number of possible flows rises rapidly.

Because any manual testing is proven to be very time-consuming and possibly error-prone, the BUC-Engine comes with a standalone tool for automated flow testing.

This tool is a standalone project, which defines an XML language for writing the BUC flow tests, which are packaged together with the BUC-Process artifacts. These tests are able to run automatically without any user involvement needed and they are well suited for integration into CI/CD systems.

For a more detailed documentation on how to write the flow tests please refer to the BUC Development guidelines documentation.

5.1 JUnit Runner

The BUC flow tests run through a custom Junit runner implementation `BucEngineTestRunner`. This runner initializes the BUC-Engine instance and provides an implementation of the in-memory datasources and an test implementation of the `DocumentSenderPlugin`. Based on the XML definition of the test flow, the runner will call the actions against this instance and perform assertions.

5.2 Flow Test Definition

A flow test definition is an XML file containing a sequence of `Steps`. The XSD for a flow test definition file is located in the `buc-engine-test` module in the file `bucTestCase.xsd`. All the steps within a single test case are performed sequentially. Following Steps are allowed:

1. `CreateCase` – instructs the test runner to create a case, e.g.

```
<createCase caseId="caseId1" bucType="aw_buc_01a"
bucVersion="{buc.model.version}" processOwnerId="BE:BE001"/>
```
2. `CaseAction` - instructs the test runner to execute an action on the case, e.g.

```
<caseAction caseId="caseId1" type="CASE_SET_PARTICIPANTS"
payloadId="participants"/>
```
3. `DocumentAction` - instructs the test runner to execute an action on the document, e.g.

```
<documentAction caseId="caseId1" type="DOC_CREATE"
documentType="DA001" documentId="DA001-1"/>
```
4. `ReceiveAction` - instructs the test runner to execute a receive message action on the case, e.g.

```
<receiveAction caseId="caseId2" receiverId="BE:BE002"
documentId="DA001-1"/>
```

5. `TimeTravel` - instructs the test runner to move forward in time to allow for assertion on timed actions, e.g.

```
<timeTravel amount="60d"/>
```

6. `AssertActionExists` - instructs the test runner perform an assertion if the a specific action exists, e.g.

```
<assertActionExists caseId="caseId1" type="DOC_CREATE"  
documentType="X001" negative="true"/>
```

A flow test case is usually a sequence of actions and assertions for the existence of actions. Currently, the test engine does not allow to perform assertions on a specific metadata objects, because these are intended to be implemented as standard unit tests of the particular components.

5.2.1 Document Content Mocking

The test engine allows to specify also a document content in the form of a JSON resource file, which can be used with the `DocumentAction` and `ReceiveAction` to simulate the reception of a real document.

5.2.2 Payload Provider

As mentioned in the previous sections, some of the actions specify a payload object to be provided when being called. The flow test engine allows to specify a payload ID, which can be chosen but should be unique for a test case definition. The test class should then provide a method annotated by `@PayloadProvider` with arguments `testId` and `payloadId`, which will be called during the test run to retrieve the payload objects to be used.

5.3 Flow Test Integration

A BUC flow test runs as a normal JUnit test. Usually the body of the test method is empty, because the test definition is provided through an annotation:

```
@RunWith(BucEngineTestRunner.class)
@BucTestConfiguration(engineConfiguration = "/test.config.properties")
@Log4j2
public class AwBuc01a_FlowTest {
    @Test
    @BucTestCase(id = "awBuc01a", file = "/aw_buc_01a_1.xml")
    public void testAwBuc01a() {
        log.info("Finished flow test - AW_BUC_01a");
    }
}
/**
 * Payload provider method, which is used to get payloads for the actions
 * from the test
 *
 * @param testId
 * @param payloadId
 * @return
```

```
*/  
@PayloadProvider  
public Object getPayloadFor(final String testId, final String payloadId) {  
    if (testId.equals("awBuc01a") && payloadId.equals("participants")) {  
        return new ArrayList<>();  
    }  
    return null;  
}  
}
```

6 Service Contract

The BUC-Engine service contract can be divided into two parts:

1. **BUC-Engine Services** – the definition is located in the module `buc-engine-api` with the implementation in the `buc-engine-core`. These services provide an interface for the communication with the engine.
2. **BUC-Process Services** – the definition as well as the implementation is located in the BUC-Processes modules. These services provide the common business logic for the BUC processes. They are versioned, hence every model version might provide a different implementation.

For more detail description of the services, please refer to the Java documentation located within the source code.

7 Data Contract

The BUC-Engine data contract can be divided into three parts:

1. **BUC Definition** – the contract is provided as a set of XSD documents located in the `buc-engine-api` module. The object model for this is generated during the build time.
2. **Metadata Model** – the metadata model represents the contract for data related to common objects like cases, documents, etc. The model is provided in the `buc-engine-api` module.
3. **Datasources and DataSourceProvider** – API for retrieving the metadata model. The interfaces are provided in the `buc-engine-api` module.

8 Exception Managements

The BUC-Engine as well as the BUC-Processes are handling any exception they can internally. However, all the unhandled exceptions are thrown to the caller. There are only couple of exception with a specific meaning.

8.1 ActionExecutionException

Any call to the BucEngine instance, which includes execution of an action may result in this type of exception to be thrown. It is up to the caller to handle this exception, however, the general rule should be that the action has failed and any transaction (if provided) should be rolled back.

In general, any non-handled exception thrown from the internal action processing is wrapped into this exception.

8.2 BusinessValidationException

Is a subclass of the ActionExecutionException and indicates, that there has been a business validation issue when executing an action. The business validations are typically encoded in the BUC-Process implementation.

The reason for this specific exception is, that the business validation exceptions might be relevant to the user. They carry also a specific message, which might be needed to be displayed in UI.

During the execution of an action every `BusinessValidationException` exception is directly thrown further to the caller. Any other exception will be wrapped into a new `ActionExecutionException`.

9 Logging Management

The BUC-Engine as well as BUC-Processes use log4j2 as the underlying logging environment. More information in “*EESSI 2020 - RINA - Architecture Overview*” [R2] document.

9.1 Logger types

Every class utilizing the Log4j2 logging uses a dedicated logger created for the fully qualified classname. Following is the description of the packages, which can be used for narrowing down the logging:

9.1.1 BUC-Engine Loggers

1. `eu.ec.dgempl.eessi.rina.buc` – base package for anything related to the BUC-Engine or BUC-Processes
2. `eu.ec.dgempl.eessi.rina.buc.core` – package containing the core classes of the BUC-Engine, i.e. the common engine processing like execution of an action, dispatching a message, etc.
 - `eu.ec.dgempl.eessi.rina.buc.core.BucEngineApplication` – logger for the BucEngine implementation
 - `eu.ec.dgempl.eessi.rina.buc.core.MessageDispatcher` – logger for the message dispatching
3. `eu.ec.dgempl.eessi.rina.buc.core.action` – package for logging anything related to the common action processing
 - `eu.ec.dgempl.eessi.rina.buc.core.action.ActionExecutor` – logger for the common action execution
4. `eu.ec.dgempl.eessi.rina.buc.core.trigger.TriggerProcessor` – logger for the common processing of triggers
5. `eu.ec.dgempl.eessi.rina.buc.api` – package containing any API classes

9.1.2 BUC-Processes Loggers

1. `eu.ec.dgempl.eessi.rina.buc.common` – package containing common BUC-Process implementation, which is used within all the processes
 - `eu.ec.dgempl.eessi.rina.buc.common.actionhandler` – package containing implemenatation of the common ActionHandlers.
 - `eu.ec.dgempl.eessi.rina.buc.common.triggerhandler` – package containing implemenatation of the common TriggerHandlers.
 - `eu.ec.dgempl.eessi.rina.buc.common.service` – package containing the implemenatation of the BUC-Process services.
2. `eu.ec.dgempl.eessi.rina.buc.<buc-type>` – package containing any custom implementation for the specific BUC type.