Logical Concept

Author(s)	Karl-Albrecht Klinge , Bolz, Gert (SMO RI R&D IXL IL) , Ghielmetti Cirillo (I-NAT-GST-CCS) , RICHTER Robert

Abstract	Describe a short abstract of the document content

Classification	public
Status	In Progress
Version	0.5
Revision	352271
Last Change Date	04.09.2024

INFO: History table is not displayed, because this document is in status **doc_inprogress**.
RULE: History table is not displayed, in statuses: { draft doc_open doc_inprogress doc_contentApproval doc_contentDecision }
CONTACT: For more information contact Administrator Disclaimer: This functional requirement specification is currently neither aligned with modeling rules defined by EET Team nor reviewed by EET Team.

Content

Contents

1 Executive Summary

Vehicles and trackside infrastructure are configured by software.
We have developed a logical concept to update them.

Use Case examples:

1. An area of control with five TPS, EAL and ATO-TS (which may collectively contain several 1000 field elements from different suppliers) needs a configuration update.
2. The ETCS vital computer and the voice radio system of a fleet of trains needs to be updated.
3. IT security patches for a lot of subsystems.
4. All ATO-TS in a country or in a corridor simultaneously need a configuration update.

Key features of our concept

- Automated configuration (updates) over the network
- Configure Systems of any size and complexity
- Manage complexity by explicit dependency management
- Use one management system for safe and non-safe configurations
- End-to-End responsibility of suppliers and integrators
- Manufacturer independent
- For vehicles and trackside infrastructure
- Process to organise suppliers, integrators and operators (RU/IM)

The EULYNX SMI interface is about transferring configuration Items to trackside field elements.
In the EULYNX concept the safety is ensured by the interlocking via the SCI interface.

We develop this concept further:

- Context in data preparation and data distribution is defined
- Documents for data distribution, dependency management and safety attestation are standardised
- For data distribution the SMI interface is needed only
- Works for vehicles and infrastructure
- Update only components that need an update

The concept is applicable for configuration updates of field elements, TPS, ATO-TS, EAL, TMS etc.

Communicating with the RU planning system and TMS an operator determines the best point in time for the configuration updates and expresses his "will" to start the update. When the interfaces to the RU planning system and TMS are specified further this process can be automated further.

2 Motivation

The new generation of railway equipment, like vehicles and TPS or ATO-TS, consists of digital components with the capability to be configured via software / data update mechanisms. This flexibility applies to the safety and the non-safety related software / data parts of these components.

This configuration management process automates the following configuration tasks, provided that the BuildingBlock is initialised to allow the communication as described in this document:

- base configuration of complex systems with explicit dependency management
- update of configurations
- initial setup of replaced BuildingBlocks

A typical configuration management process relies on different stakeholders and responsibilities whereby suppliers, integrators and operators (RU/IM) are the main actors.

To ensure that these components work together safely and effectively in systems of arbitrary size and complexity, there is the need to define and standardise the configuration management process.

This process must define explicit configuration management coordination principles. This requires (at least) standardised meta-information for automated distribution and orchestration of artifacts, verification, and a validation process. This configuration management process must ensure the integrity and authenticity of all configuration items (artifacts) and their dependencies. It is important that the process allows each supplier and integrator to take over responsibility in line with their competence.

For example:

We have an area of control with three TPS, EAL and ATO-TS (which may collectively contain 1000 field elements from different suppliers) which requires a software configuration update.

Without explicit, safe management of dependencies and automated distribution, this is hard to manage, error prone and may result in a misconfiguration. The same applies to a vehicle with all its components (e.g., from the safe computer to door systems) that could be in different hardware revisions in different homologated versions within one train series.

3 Purpose

Purpose of this document is to create abstract models and principles that help in reasoning, designing, and problem-solving for software / data configuration of dependent safe and non-safe subsystems.

This involves the development of logical structures, algorithms, data models, and a theoretical framework.

Before reading this document, it is recommended to read the context and the basic concepts that drive the CCS configuration management process provided here:

- SD3 Operational Analysis (ORS)
- SD3 System Analysis (FRS)

4 Glossary

4.1 Terms and definitions

Title	Description
Building Block	A Building Block is an equipment based (hardware and/or software) or logical unit of the System having: • standardised functionality or aggregates standard functionality it depends on • may have standardised PRAMS requirements (including Tolerable Functional Failure Rate [TFFR] • may have Safety Integrity Levels [SIL] for functions within the system border and Safety Related Application Conditions [SRAC]) • standardised cyber security requirements (including Security Level [SL] based on the security requirements, and Security Related Application Conditions [SRAC]) • may have (on lower levels) standardised interfaces (on all OSI Layers) towards other Building Blocks and/or external systems. Equipment based Building Blocks are separately sourceable from different suppliers and capable of being integrated by a third party (integrator).

	A BuildingBlock has one or more BuildingBlockConfigurations. A BuildingBlock must have a unique identifier composed of configurationGroupId and configurationId.
Building Block Configuration	A BuildingBlockConfiguration (BBC) is a node on a layer within the configuration dependency tree. It must be uniquely identifiable within the system and may contain a configurationFile artifact and dependencies to other BBCs. One BuildingBlock (BB) can have one or more BuildingBlock Configurations (BBC). One Building BlockConfiguration (BBC) has exactly one configuration.json file (and a configurationSafe.json if it is a safe BBC). BBCs that itself have no further dependencies in their configuration.json file are the Lowest Updatable Units (LUU - can be updated on its own). BBCs that are updatable must provide a corresponding configurationFile (payload). BBCs that are updatable need an endpoint described in the "configuration.json" file. That BBC endpoint can be accessed using a protocol capable of file transfer (e.g. opc ua).
BuildingBlockConfiguration "configuration.json" document	The BuildingBlockConfiguration configuration.json document describes a BuildingBlock Configuration. In the section <dependencies> the file lists all BuildingBlockConfiguration configuration files the BuildingBlockConfiguration depends on. The BuildingBlockConfiguration "configuration.json" document references other BBC configuration.json documents recursively. This forms a dependency tree that is resolved recursively until the BBC "configuration.json" has no further dependencies. These are the LUU - Lowest updatable units. The LUUs link a configurationFile (payload). As an example there may be different BBC "configuration.json" documents for a specific BuildingBlock: A generic BuildingBlockConfiguration including a firmware as "generic definition" (before integration in its environment) that has been compiled by the BuildingBlock supplier. There might be another BuildingBlockConfiguration from the integrator that includes the parametrization of the generic product and depends on the correct firmware BBC. The BBC configuration is defined within a "configuration.json" document that is validated by the configuration.schema.json.
CCS Deployment	CCS Deployment refers to one physical deployment of a CCS System, that is uniquely identifiable with configurationGroupId, configurationId and configurationVersion. A CCS Deployment consists of the CCS hardware running the BBCs that are defined in a Top-Level BuildingBlockConfiguration (BBC) and its dependencies.

Configuration Management	Configuration Management refers to the management of all Configuration Items of a physical instance of a System (e.g. trains or trackside CCS). From a process point of view, it covers activities along the complete chain, from the Building Block Supplier(s) to the Integrators, the Operator, and the actual Deployment being operational.
Configuration Management Concept	The Configuration Management Concept is a solution oriented description of the Configuration Management Process: it defines all the technical solutions needed for the concrete implementation of the Configuration Management Process.
Configuration Management Process	The Configuration Management Process captures all concrete activities along the complete configuration management chain: from the Building Block realisation (by the Building Block Supplier(s)), to the integration into a specific System Instance (by the Integrator(s)), to the distribution to the field (by the Operator), and the actual activation (going into operation) in the field (it includes also on a train). The Configuration Management Process is a solution agnostic term that collects all activities needed for the Configuration Management.
DistributionJob "distribution-job.json" document	A DistributionJob is defines the time and conditions when one or more CCS deployments with the same homologation receive their software configuration update (preload and activation). Examples for conditions when the distribution-job is started are approvals from a train-driver, or someone from operation or a GOA4 system that need to approve the configuration update within the preloading and activation time windows. The distribution-job.json document references the Top Level BuildingBlockConfiguration (BBC), that has recursive dependency tree. The distribution-job is defined within a distribution-job.json document that is validated by the distribution-job.schema.json.
Hardware	Hardware refers to the physical components of a computer system or electronic device. It encompasses all the tangible parts that can be seen and touched, such as the central processing unit (CPU), memory modules, storage devices (hard drive, solid-state drive), possible input/output devices (e.g. keyboard, mouse, monitor, etc), motherboard, graphics card, sound card, and various other components that make up a computation unit or electronic device. Hardware is responsible for executing and processing instructions and data, providing the necessary functionality for software to run and perform tasks.

High level BuildingBlocks	Due to the dependency concept the BuildingBlocks can form units of unlimited size. The recursive dependency tree can have any depth. Every level of the dependency tree links to the next level - that allows to assign clear responsibilities for each of the levels. As an example a BuildingBlock "area" might refer to a number of "interlocking interiors" and "filedelements" in its next downstream dependency level. The Top-Level BuildingBlockConfiguration is the root where all dependencies start from.
Operator	An institution operating a system through its life-cycle, in context of the document usually an Infrastructure Manager or Train Operator.
Service Function Configuration	The Service Function Configuration (SFC) is the implementation of the Configuration Management System. The SFC is a central location technical system that is responsible for managing the BuildingBlock Configurations. Each BuildingBlock deployment is managed by exactly one SFC.
Software	Software is software that encompasses both : the operating system and the computer application that runs the computer. They are both essential for the operation of a device.
Static (or semi-static) data	Static (or semi-static) data refers to information or data that remains unchanged or constant over time. It is data that does not require frequent updates or modifications. Static data typically includes reference data, constants, configuration settings, or any other data that remains consistent throughout the operation of a system or application. This type of data is often used as a foundation or reference point for various processes or calculations within a system. Static data is generally stored in a read-only format and is not subject to frequent modifications or user interactions. This is often data that requires a homologation process before it can be applied on a system going in operation. However, it can also include data that does not require a homologation like IP-address, security patches, etc. Example of statical data: software, firmware, parametrisation file, data related to the topology stored in an interlocking, braking curves stored in the ETCS on-board, etc.

14 items found

4.2 Abbreviations

Abbreviation	Title
BB	Building Block
HW	Hardware
SFC	Service Function Configuration
SW	Software

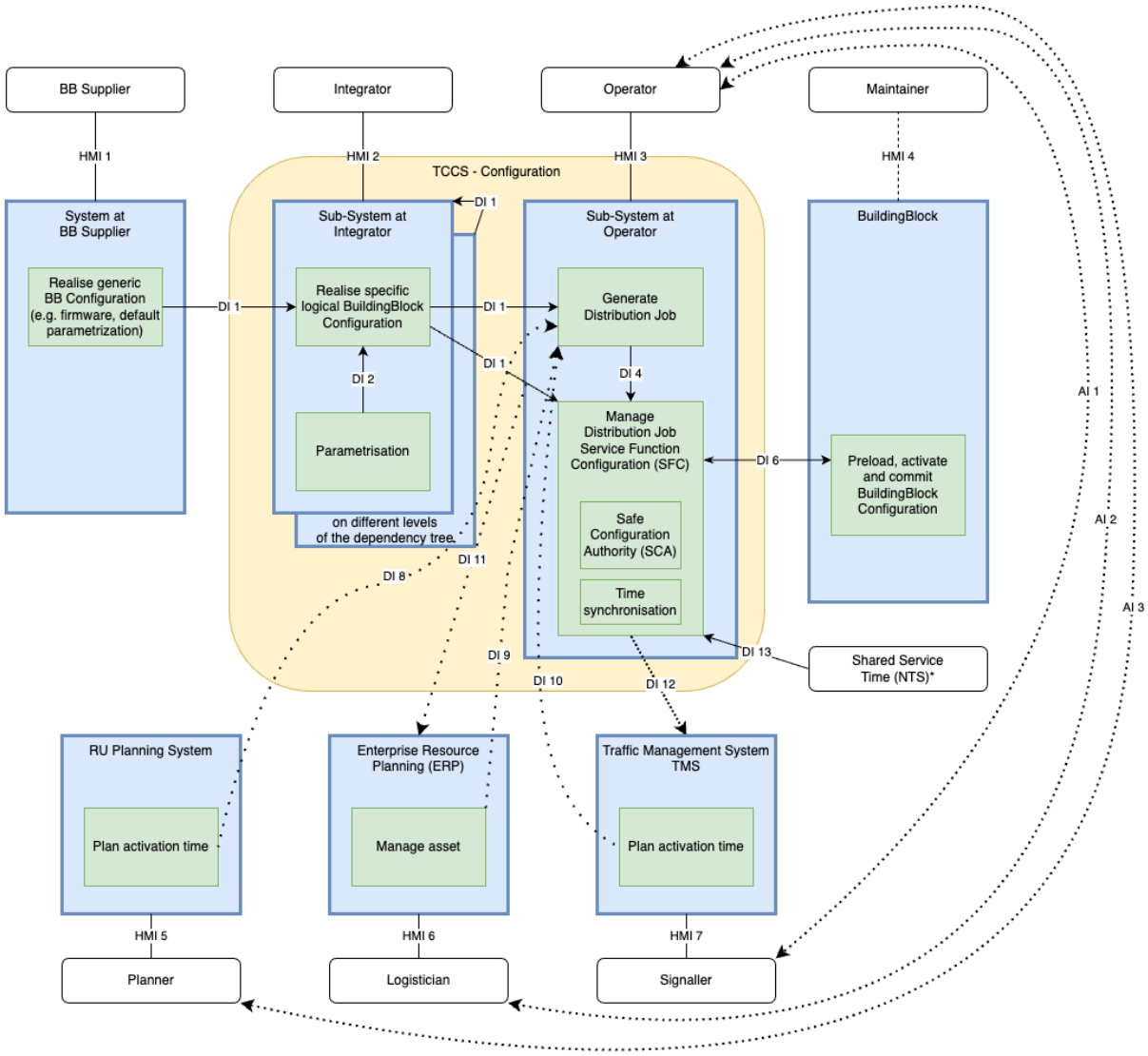
4 items found

5 Solution concept and principles

5.1 Environment and Systems for Configuration Management

SPT2TS-125572 - Environment and systems for the configuration management process

The following figure shows the environment and the different systems involved in the configuration management process including their actors (stakeholders):



System	Description	Function
System at Building Block (BB) Supplier	From configuration management perspective, this is the system used by the BB Supplier as interface to the Integrator. It is used by the BB Supplier to publish generic BuildingBlockConfigurations. Some of these generic BuildingBlockConfigurations will remain unchanged until installation (typically executable like software, firmware, etc.). The supplier will also create default parametrisation files whose content is adjusted to the specific logical application by an Integrator. The supplier will provide the limits for all the parameter values. This includes a repository to be used to distribute	The system publishes (binary) configurationFile items and additional files identifying the BuildingBlockConfiguration (BBC) to be accessed in a repository (see below).

	both safety relevant and non-safety relevant BuildingBlockConfigurations (interface **DI 1**). DI 1 is the access to a repository that stores BBC artifacts. That is applicable for BBC artifacts that a supplier provides to an integrator, integrators to integrators or integrators to operators.	
(Sub-)system at Integrator	From configuration management perspective, this is the data preparation subsystem used by the Integrators on different levels of the system. On the lower levels it depends on the generic BuildingBlockConfigurations published in the repositories of the BuildingBlock Suppliers and defines specific BuildingBlockConfigurations that contain the parametrisation for an instance of a logical railway (sub-)system. On higher levels the Integrators create logical BuildingBlockConfigurations that describe the composition of BuildingBlockConfigurations one level down. The number of levels is not restricted. As an example for trackside: if an area of control is described that contains a number of TPS, EAL and ATO-TS to be configured as a combined system: BBC area (==Top Level BBC) (depends on a number of) BBC interlocking (depends on a number of) field element logical parametrization (depends on) field element firmware The Integrators on different levels of the system need not necessarily be in one company. The interface (**DI 1**) between the different levels is a BuildingBlockConfiguration repository. The BuildingBlockConfigurations form a dependency tree (see below). It is useful to differentiate between safe and non-safe BuildingBlockConfigurations to reduce the effort for non-safe configuration updates. For this reason the integrity of safe BBCs is secured by a sidecar file that contains hashes of the dependencies one level down recursively. *We do not define the binary configurationFile that contains e.g. firmware or parametrization. If data models are needed e.g. for the parametrization of the interlocking topology we assume that the SD1 data models will be used.*	The following functional range is provided: • Management of (generic) BuildingBlockConfigurations published by BuildingBlock Suppliers. • Management of BuildingBlockConfigurations (BBC) that are dependent on other BBCs, e.g. a parametrization BBC dependent on the generic BBCs of the supplier including the logical parametrisation files with parametrisation values adjusted to the specific application. • Produce the configuration.json and configurationSafe.json (see below) to describe the dependencies on exact versions of BBCs one level down. • Publish the Top-Level BuildingBlockConfigurations including all BuildingBlockConfigurations in a repository accessible by the Operator.

Sub-system at Operator	From configuration management perspective, this is the system used by the Operator to select the Top-Level BuildingBlockConfiguration provided in the Integrator(s) repositories (DI 1) and define which target shall when preload and activate a BuildingBlockConfiguration dependency tree based on the distribution-job provided in DI 4. The ServiceFunctionConfiguration (SFC) stores which BuildingBlock is using which exact BuildingBlockConfiguration (version). It is also capable of analysing and comparing different versions of the BuildingBlockConfiguration dependency trees to know which BuildingBlockConfiguration needs to be updated on a Top-Level BuildingBlockConfiguration version switch. It manages the activation and Confirmation of all BuildingBlockConfigurations that need a version change. For the confirmation of safe BuildingBlockConfigurations it contains an addon function called SafeConfigurationAuthority (SCA). For asset management in an Enterprise Resource Management (ERP) system it could be helpful to offer an interface to read the current BuildingBlockConfiguration versions from this system.	The following functional range is provided: • Management of Top-Level BuildingBlockConfigurations published by Integrators. • Producing the distribution-job defining the target and when to preload and activate a safe/non-safe BuildingBlockConfiguration. • Process the distribution-jobs to update the BuildingBlocks (target components).
Building Block	From configuration management perspective, this is the target of the single configuration items. This component is commanded to preload and activate the BuildingBlockConfigurations needed. This component reports back the status of preloading and activation.	The BuildingBlock preloads and activates new BuildingBlockConfigurations as commanded.
RU Planning System	From configuration management perspective, the operator recognises, based on the information provided by this system, the appropriate point in time for the activation of the BuildingBlockConfigurations on vehicles. The information is used for the definition of the distribution-job. From the data provided by the RU Planning System the operator identifies when and for which period a specific train is not planned for operation.	The system provides the operational planning information of relevant on-board instances.
Enterprise Resource Planning (ERP) - optional	The ERP system can read the current BuildingBlockConfigurations installed on a physical BuildingBlock for asset management purposes. It may link this information to the equipment model (see equipment model in SDI).	optional - read only to Sub-system at Operator

Traffic Management System (TMS)	From configuration management perspective, the operator recognises, based on the information provided by this system, the appropriate point in time for the activation of trackside BuildingBlockConfigurations. The information is used for the definition of the distribution-job. From the data provided by the TMS Management System the operator identifies when and for which period specific trackside equipment (a specific track area) is not planned for operational activities. TMS should also be informed about non-availabilities due to configuration changes.	The system provides the operational planning information of relevant trackside logical and physical instances.

Table 1 Description of the different involved systems

Interface ID	Description	Exchanged data
HMI 1	Human Machine Interface 1: Interface between the human BB supplier and the system at BB supplier. *This interface is proprietary to the specific BB supplier and will not be addressed by Transversal CCS domain now. It could be useful to standardise some aspects later to reduce training costs.*	Information about the contents of available BuildingBlockConfigurations in the suppliers repository.
HMI 2	Human Machine Interface 2: Interface between the human integrator and the system at integrator. *This interface is proprietary and will not be addressed by Transversal CCS domain now. It could be useful to standardise some aspects later to reduce training costs.*	Information relevant for the following activities: • Parametrisation of BuildingBlockConfigurations. • Ensure integrity of the dependencies of different BuildingBlockConfigurations between each other on different aggregation levels. • Publication of BuildingBlockConfigurations in a repository.
HMI 3	Human Machine Interface 3: Interface between the human operator and the system at operator. *This interface is proprietary to the specific operator and will not be addressed by Transversal CCS domain*	Information relevant for the following activities: • Creation of distribution-job. • Show valid distribution-jobs • Select a valid distribution-job for execution ("will" of the Operator).

	now. It could be useful to standardise some aspects later to reduce training costs.	• Visualise the execution of a distribution-job: which BuildingBlockConfiguration versions need to change; preload status, the activation and commit status of BuildingBlockConfigurations on BuildingBlocks • Visualise the BuildingBlockConfiguration version history including all state changes (e.g. PreloadStatus and ActivationStatus).
HMI 4	Human Machine Interface 4: Interface between the human Maintainer and the BuildingBlock. *This interface is proprietary to the specific BuildingBlock and will not be addressed by Transversal CCS domain now. It could be useful to standardise some aspects later to reduce training costs.*	Information relevant for maintenance activities.
HMI 5	Human Machine Interface 5: Interface between the human planner and the RU planning system. *This interface is proprietary to the specific RU planning system and will not be addressed by Transversal CCS domain now. It could be useful to standardise some aspects later to reduce training costs.*	Information relevant for the operational planning of the different trains.
HMI 6	Human Machine Interface 6: Interface between the human logistician and the enterprise resource planning (ERP). *This interface is proprietary to the specific enterprise resource planning (ERP) and will not be addressed by Transversal CCS domain now.*	Information relevant for the management of the CCS asset.
HMI 7	Human Machine Interface 7: Interface between the human signaler and the traffic management system (TMS). *This interface is to a certain extent addressed by the SD4 team of Transversal CCS domain. The SD3 team (configuration management process) will not look to the details of this interface now.*	Information relevant for the traffic planning of the area of control covered by the specific system.

DI 1	Digital Interface 1: This interface is used to publish and provide the BuildingBlockConfigurations to Integrators and the Operator (repository). It will use the identical structure on all levels of the BuildingBlockConfiguration dependency tree. *Comment: If e.g. data from a Digital Register need to be transferred in a specific version the respective integrator will create a BuildingBlockConfiguration (a binary configurationFile including the contents like the segment profiles, the configuration.json and configurationSafe.json files according to the chapters below). That BuildingBlockConfiguration is transferred to a Trackside Protection System as a BuildingBlock according to the processes described below.* **This interface must be standardised .** **There are many solutions already available - we need to choose one of the standard repository interfaces.**	A repository URI will contain: • configuration.json • configurationSafe.json • configurationFile (binary, optional) • hashes and signing information See below for further information.
DI 2	Digital Interface 2: This interface is used to get the information needed to create a specific BuildingBlockConfiguration for a logical application. On this basis the Integrator can define the parametrisation values within the boundaries defined by the BuildingBlock supplier. *This interface is proprietary between BuildingBlock Supplier and Integrator and will not be standardised.*	BuildingBlock supplier provides a BuildingBlockConfiguration parametrization template including boundaries and default values to be applied by Integrator. This depends on the version of the BuildingBlockConfiguration to be parametrised.
DI 4	Digital Interface 4: This interface is used at the operator to release a distribution-job that contains the specific targets and the preloading and activation information. The distribution-job will only link the BuildingBlockConfigurations provided in the DI 1 interfaces. **This interface must be standardised.** **It could be a good solution to use the DI 1 repository for the distribution-job also.**	
DI 6	Digital Interface 6: This interface is used by the SFC to preload, activate and confirm BuildingBlockConfigurations on physical BuildingBlocks. The preload, activation and confirmation states must	configurationJson configurationFile (binary, optional) hashes preload, activation and conformation states

	be available for visualisation to the operator during an update and must also be historised. **This interface must be standardised.**	see below for more details including sequence diagrams.
DI 8	Digital Interface 8: The information transferred over this interface is used by the operator to adjust the distribution-job to the operational plans of the single affected train units. Operator needs to ensure that a specific train is not in operation when BuildingBlockConfigurations are activated. *Task: at first there is no intention to standardise this interface as it has a lower priority compared to other interfaces involved in the configuration management process. Furthermore, the involved systems can be technologically quite different between different operators and planners. The workaround is to use "AI 3".* *At a later date this decision can be revised.*	Information relevant for defining the distribution- job of CCS on-board instances.
DI 9	Digital Interface 9: Optional: the asset management system can read which BuildingBlockConfigurations are activated on which physical BuildingBlock. Task in SD3: at first there is no intention to standardise this interface as it has a lower priority compared to other interfaces involved in the configuration management process. Furthermore, the involved systems can be technologically quite different between different operators and logisticians. The workaround is to use "AI 2". At a later date this decision can be revised.	needed for seamless integration of asset management systems (ERP)
DI 10	Digital Interface 10: This interface is used by the operator to adjust the distribution-job to the operational plans of affected tracks. Operator needs to ensure that tracks affected by the BuildingBlockConfiguration updates are not planned for operational functions when BuildingBlockConfigurations is activated. *Task in SD3: at first there is no intention to standardise this interface as it has a lower priority compared to other interfaces involved in the configuration management process. Furthermore, the involved systems can be technologically quite different between different operators and signalers. The workaround is to use "AI 1".* *At a later date this decision can be revised.*	Information relevant for defining the distribution-job of CCS trackside instances.

DI 11	Digital Interface 11: This interface is used by the logistician to update the enterprise resource planning (ERP) system based on the definition and installation result of the specific BuildingBlockConfigurations of a BuildingBlock. *Task in SD3: currently there is no intention to standardise this interface as it has a lower priority compared to other interfaces involved in the configuration management process. Furthermore, the involved systems can be technologically quite different between different operators and logisticians. The workaround is to use "AI 2".* *At a later date this decision can be revised.*	Information relevant for managing the enterprise resource planning (ERP) system.
DI 12	Digital Interface 12: This interface is used by the operator to distribute and publish BuildingBlockConfiguration dependency tree and the distribution-job information to the traffic management system (TMS). This interface is used to ensure alignment of topology data between the trackside BuildingBlockConfigurations and the TMS *Task in SD3: currently the configurationFile (binary) that contains such information is not standardised. Probably it could be a solution to treat the TMS to be a BuildingBlock and use the same interface as DI 6.*	Topology data information that is used by CCS Building Blocks as well as by the TMS.
DI 13	Digital Interface 13: Process the distribution-jobs to update the BuildingBlocks (target components). The time will finally be used for activation of a CCS configuration. *Please note that for security and certificate validation the time service is required at all configuration management systems.* *This topic is also addressed by the Security domain that might already standardise this interface as a shared service.*	Time synchronisation information, being typically a time synchronisation protocol.
AI 1	Analogue Interface 1: This interface is used by the operator to synchronise with the signaller the operational plans of tracks affected by the installation of a CCS configuration. The interface is an exchange that can be in person, verbally by telephone or by other means. In the future this interface might be replaced by "DI	Information relevant for defining the distribution-job (activation information) of CCS trackside instances.

	10".	
AI 2	Analogue Interface 2: It can be used by the logistician to update the ERP based on the activation state of the BuildingBlockConfigurations on physical BuildingBlocks. The interface is an exchange that can be in person, verbally by telephone or by other means. In the future this interface might be replaced by "DI 9" and "DI 11".	Information relevant for managing the enterprise resource planning (ERP) system.
AI 3	Analogue Interface 3: This interface is used by the operator to synchronise with the planner the operational plans of single train units affected by the BuildingBlockConfiguration activations. The interface is an exchange that can be in person, verbally by telephone or by other means. In the future this interface might be replaced by "DI 8".	Information relevant for defining the distribution-job (activation information) of CCS on-board instances.

Table 2 Description of the different interfaces

5.2 Configuration Data Preparation

SPT2TS-124931 - Data preparation artifacts

During data preparation suppliers and integrators are authoring "configuration.json" documents on different levels of the dependency tree (WHAT).
The distribution of these items is described in the "distribution-job.json" documents (WHEN).

These documents are described in this chapter.

SPT2TS-126690 - Safe BuildingBlockConfiguration

Some of the BBC configuration artifacts that are prepared during data preparation are safe. The authoring entity must make sure that the preparation process of these artifacts meets the safety requirements according to the SIL level of the functions configured by the BuildingBlockConfiguration.

5.2.1 Configuration Metadata

SPT2TS-124929 - BuildingBlockConfiguration "configuration.json" document

BuildingBlockConfigurations "configuration.json" documents are an exhaustive, unambiguous human and machine readable description of a BuildingBlockConfiguration (BBC) approved by a responsible entity, defined

by a standardised lightweight data-interchange format.

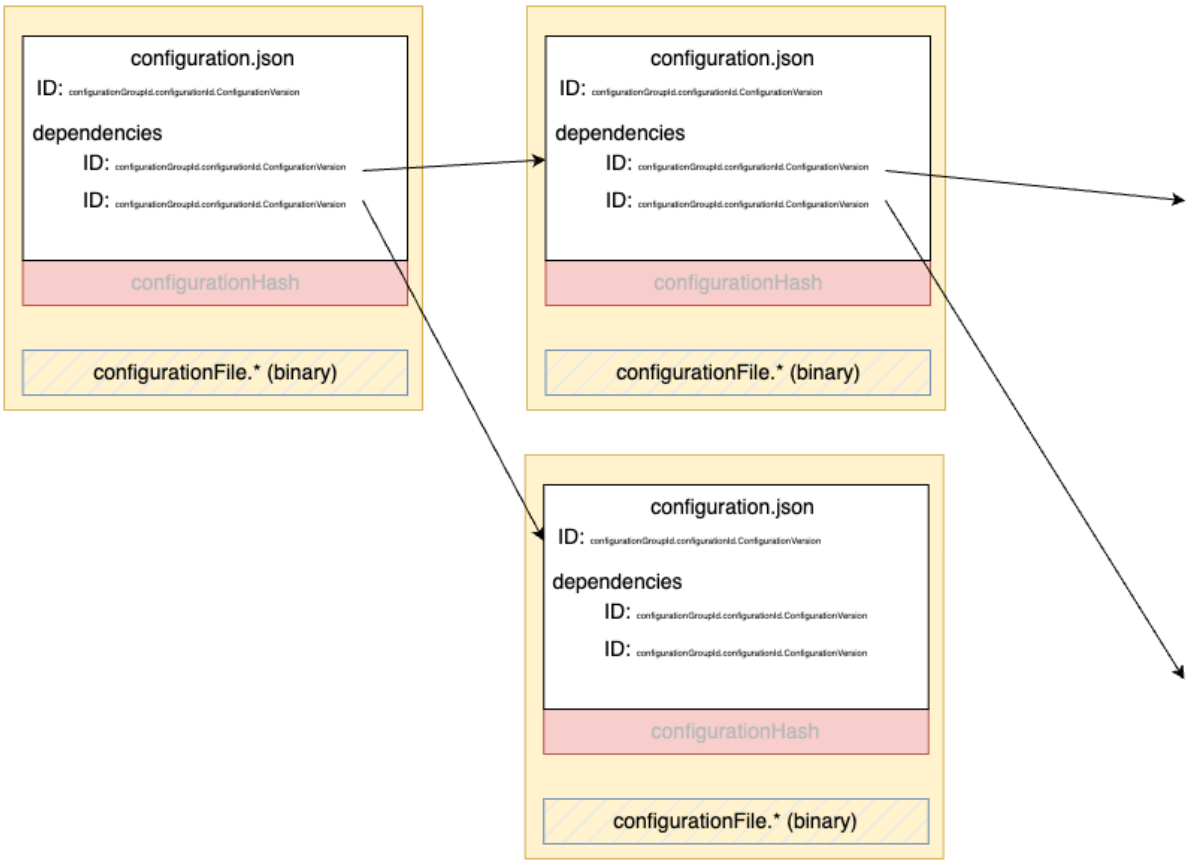
SPT2TS-126685 - Specific to logical components and/or hardware models

BuildingBlockConfigurations are specific to logical components (e.g. a subsystemId) and/or hardware-models. BuildingBlockConfigurations must not be hardware specific (i.e. for a specific serial number).

Reason: if configurations would be hardware specific (linked to the specific serial number) integrators have to prepare an individual config for each vehicle or create a new one in case of a component failure where a component needs to be replaced. This would cause long downtimes, because all upper integrator levels have to check and sign.

SPT2TS-126686 - Concept of Dependencies

The BuildingBlockConfiguration "configuration.json" document uses dependencies. The dependencies are resolved recursively and build a tree structure from the Top-Level BuildingBlockConfiguration that recursively depends on other BuildingBlockConfigurations. The number of dependency levels is not limited.



This dependency tree structure allows to built greater entities.

The concept also allows for having multiple levels of responsibilities.

A supplier is end-to-end responsible the correctness of his "configuration.json" document that in most cases will also reference a binary "configurationFile.*" including e.g. a firmware.

An integrator is end-to-end responsible the correctness of his "configuration.json" document that in the lower part of the dependency tree will also reference a binary "configurationFile.*" including e.g. a parametrization.

A "configuration.json" document of an integrator can be identified by the existence of "dependencies". Integrators can exist on multiple levels. An integrator is end-to-end responsible for the contents of his "configuration.json" document that must also include the correct composition of the included dependencies.

SPT2TS-126689 - Identification of BuildingBlockConfiguration on different levels of the dependency tree

A BuildingBlockConfiguration is uniquely identified by a triple of attributes:

1. configurationGroupId: e.g. according to com.supplier-b.productfamily-a.productname-b
2. configurationId: e.g. according to model-c.firmware
3. configurationVersion: according to major.minor.patch according to the semantic versioning standard 1.0.0 e.g. 1.0.3

SPT2TS-124962 - Building BlockConfiguration "configuration.json" documents

The BuildingBlockConfiguration (BBC) "configuration.json" document describes one BuildingBlockConfiguration. One BuildingBlock can have multiple BuildingBlockConfigurations.

The information contained in a "configuration.json" document must be transparent.

It may link a configurationFile (payload) of different types: e.g. software or parameter packages.

The configuration file must contain information to uniquely identify the BuildingBlockConfiguration item by configurationGroupId, configurationId and configurationVersion.

SPT2TS-125852 - BuildingBlockConfiguration "configuration.json" syntax

The BuildingBlockConfiguration (BBC) "configuration.json" uses the json syntax.

A json-schema document is provided to describe the syntax.

BuildingBlockConfigurations must comply to this configuration.schema.json document.

SPT2TS-125986 - Semantics of the attributes of the BuildingBlockConfiguration "configuration.json" document

Attribute	Description
configurationGroupId	The configurationGroupId must be generally unique amongst an organization and a group of BuildingBlockConfigurations. Examples according to: com.supplier-b.productfamily-a.productname-b (supplier product) com.integrator-a (integrator)
configurationId	The configurationId is generally the name that the BuildingBlock is known by. This name may be a "model" on lower levels of the dependency tree or different aggregations of subsystems on higher levels of the dependency tree like

	subsystemIds or interlocking names or the name of a vehicle fleet whose vehicles have the identical homologation. Examples according to: model-c.firmware (device specification on lowest level) logical.point.subsystem-0815.param (logical (sub-)system specification) logical.interlocking.paris-nord.param (logical (sub-)system specification on higher levels) It, along with the configurationGroupId, creates a key that separates this BuildingBlockConfiguration from every other BuildingBlockConfiguration in the world. Along with the configurationGroupId, the configurationId fully defines the configuration's living quarters within the repository (see repository).
configurationVersion	The version denotes which specific incarnation of the BuildingBlockConfiguration denoted by configurationGroupId.configurationId is meant. For the version strings the Semantic Versioning 1.0.0 specification must be used: major.minor.patch. The version string also contains "SAFE" for a safe BuildingBlockConfiguration and "NONSAFE" for a non-safe BuildingBlockConfiguration. Examples: 4.12.0-SAFE 3.8.2-NONSAFE
configurationFileName	The name of the (optional) configuration file in the same directory or URL path next to the "configuration.json" file. Example: parametrization.zip
configurationFileHash	The fingerprint of the optional configurationFile. If a configurationFile exists this is a mandatory attribute. This fingerprint must include the logical id (e.g. the subsystemId) and the hardware model. It must NOT include a fingerprint of an individual hardware (e.g. a serial number). If we would allow more specifics in the configurationHash we would not be able to: • exchange a hardware equipment with another hardware of the same model without updating the safe hashes on all higher levels of the dependency tree • distribute the configuration to entities of the same model (e.g. to all vehicles of a fleet that have the same model). In this case an individual integration process must be made for each vehicle of the fleet.
configurationFile SignatureAlgorithm	"ecdsa-with-sha512" must be used. (see SPPRAMSS-7356 - The Manufacturer Config Signer Certificate (MCSC) shall fulfill the certificate...)

configurationFile Signature	base64 encoded signature
configurationFileTarget	URI of the target for file transfer including the protocol stack: e.g. "opcua:4840://subsystem0815.intern.dstwxy.com/import/parametrization" The information of this variable is used by the SFC. It allows defining to which destination (target) a specific configuration item shall be sent during the preload. Same information is also used during the activation, when SFC is communicating with the destination instance for the activation.
dependencies	An important part for a BuildingBlockConfiguration is its dependency list. Most BuildingBlockConfigurations depend on others to update and run correctly. The list of dependencies are resolved recursively to the dependent BuildingBlockConfigurations. The recursive resolution of BuildingBlock dependencies stops when BuildingBlockConfigurations do not contain dependencies themselves. Dependencies on the lower levels must be installed first (bottom-up). E.g. a firmware must be activated before the parametrization that has this firmware version in its dependencies. This mechanism allows multi-level configuration management: each level can focus on the direct (one level down) BuildingBlockConfiguration dependencies. If we change the top level version, all dependencies will resolve automatically. Dependencies are identified by the same attributes ***configurationGroupId.configurationId.configurationVersion*** as the "configuration.json" document that contains them. The dependency references a BuildingBlockConfiguration "configuration.json" document. Here is an example of one entry of a list of dependencies: `"dependencies": [{ "configurationGroupId": "com.supplier-b.productfamily-a.productname-b", "configurationId": "model-c.firmware.safe", "configurationVersion": "2.0.6", "configurationSafetyRelevance" : "SAFE" }, { "configurationGroupId": "com.supplier-b.productfamily-a.productname-b", "configurationId": "model-c.firmware.nonsafe", "configurationVersion": "1.0.3", "configurationSafetyRelevance" : "NONSAFE" }]`
repositories	Here is an example of a list of repositories: `"repositories": [ { "sequenceNr": "0", "name": "Main Repository", "url": "https://repository.integrator-a.com" },`

	<pre>{
	"sequenceNr": "1",
	"name": "Backup Repository",
	"url": "https://backup.integrator-a.com"
	}
	]</pre>
	All the dependencies must be found in one of the repositories listed.
	
	The repository is identified by its "url" and gets a human readable "name". The "url" defines where to find the repository - for example a REST api service accessible via https (see example above).
	
	The optional "sequenceNr" indicates the order in which repositories are requested for the artifacts (starting with lowest number going upwards).

Table 3 BuildingBlockConfiguration "configuration.json" description

SPT2TS-127127 - configuration.schema.json

Please find the schema as an attachment in the "Work Item Properties" of this item.

The configuration.json uses the json syntax. A json-schema document is provided to describe the syntax. configuration.json documents must comply to this configuration.schema.json:


```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "array",
  "items": [
    {
      "description": "Configuration document.",
      "type": "object",
      "properties": {
        "@schemaLocation": {
          "type": "string",
          "const": "http://schemas.rail-research.europa.eu/configuration/V0.1"
        },
        "modelVersion": {
          "type": "string"
        },
        "configurationGroupId": {
          "type": "string"
        },
        "configurationId": {
          "type": "string"
        },
        "configurationVersion": {
          "type": "string"
        },
        "configurationSafetyRelevance": {
          "type": "string",
          "enum": ["SAFE", "NONSAFE"]
        },
        "configurationFile": {
          "type": "object",
          "properties": {
            "configurationFileName": {
              "type": "string"
            },
            "configurationFileHash": {
```

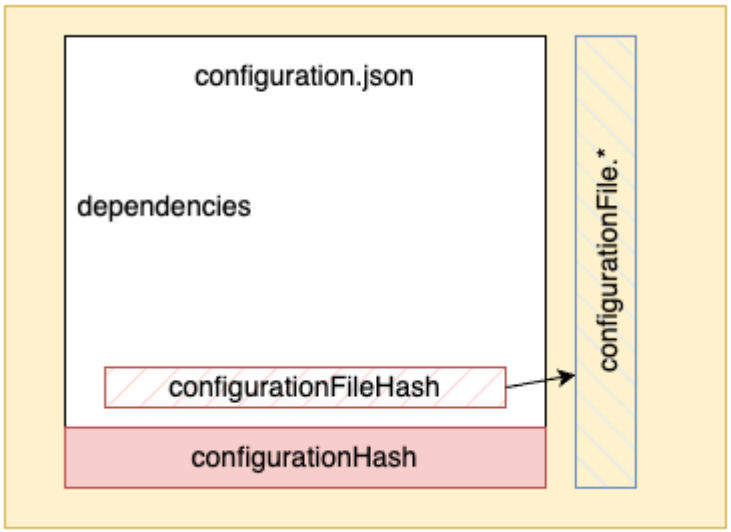
```
        "type": "string"
      },
      "configurationFileSignatureAlgorithm": {
        "type": "string"
      },
      "configurationFileSignature": {
        "type": "string"
      },
      "configurationFileTransferProtocol": {
        "type": "string"
      },
      "configurationFileTransferHostPath": {
        "type": "string"
      }
    },
    "required": [
      "configurationFileName",
      "configurationFileHash",
      "configurationFileSignatureAlgorithm",
      "configurationFileSignature",
      "configurationFileTransferProtocol",
      "configurationFileTransferHostPath"
    ]
  },
  "name": {
    "type": "string"
  },
  "url": {
    "type": "string",
    "format": "uri"
  },
  "dependencies": {
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "configurationGroupId": {
          "type": "string"
        },
        "configurationId": {
          "type": "string"
        },
        "configurationVersion": {
          "type": "string"
        },
        "configurationSafetyRelevance": {
          "type": "string",
          "enum": ["SAFE", "NONSAFE"]
        }
      },
      "required": [
        "configurationGroupId",
        "configurationId",
        "configurationVersion",
        "configurationSafetyRelevance"
      ]
    }
  },
  "repositories": {
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "name": {
```

```
                "type": "string"
              },
              "url": {
                "type": "string",
                "format": "uri"
              },
              "sequenceNr": {
                "type": "integer"
              }
            },
            "required": ["name", "url"]
          }
        }
      },
      "required": [
        "@schemaLocation",
        "modelVersion",
        "configurationGroupId",
        "configurationId",
        "configurationVersion",
        "configurationSafetyRelevance",
        "name",
        "url",
        "repositories"
      ]
    },
    {
      "description": "The attribute 'configurationHash' in this object contains a hash over the first object, providing an integrity check to ensure that the primary configuration has not been altered.",
      "type": "object",
      "properties": {
        "@schemaLocation": {
          "type": "string",
          "const": "http://schemas.rail-research.europa.eu/configurationHash/V0.1"
        },
        "configurationHash": {
          "type": "string"
        }
      },
      "required": [
        "@schemaLocation",
        "configurationHash"
      ]
    },
    {
      "description": "integrator addon to define the configurationFileTransferHost.",
      "type": "object",
      "properties": {
        "configurationFileTransferHost": {
          "type": "string"
        },
        "configurationFileTransferHostHash": {
          "type": "string"
        }
      },
      "required": [
        "configurationFileTransferHost",
        "configurationFileTransferHostHash"]
    }
  ],
  "minItems": 2,
  "maxItems": 3
}
```

SPT2TS-125640 - Ensuring the integrity of the "configuration.json" file and its dependencies

The integrity of a BuildingBlockConfiguration "configuration.json" file is ensured by the configurationHash that is stored as as a separate object within the "configuration.json" file.

Remark: This could also be solved with a separate file "configuration.json.sha1" in the same directory or URL path.



If the configuration.json file references a "configurationFile.*", then the configuration.json file must also include a configurationFileHash that is calculated over the "configurationFile.*". The configurationFile.* must also reside in the same directory or URL path next to the configuration.json file.

Comparison to security-team concept:
Dependencies are currently not in scope of security that looks at one Lowest Updatable Unit (LUU), see SPPRAMSS-7342 - figure: .
The process described in security including the operator as an entity that decrypts and encrypts the manifests and files again is different to the SD3 concept, please compare SPPRAMSS-5794 . In SD3 the operator is responsible for the time of the update, but the integration is taken by a role "integrator" that does not exist in the concept from the security team.

SPT2TS-127435 - Taking Out of Operation (Top-Down in the Dependency Tree)

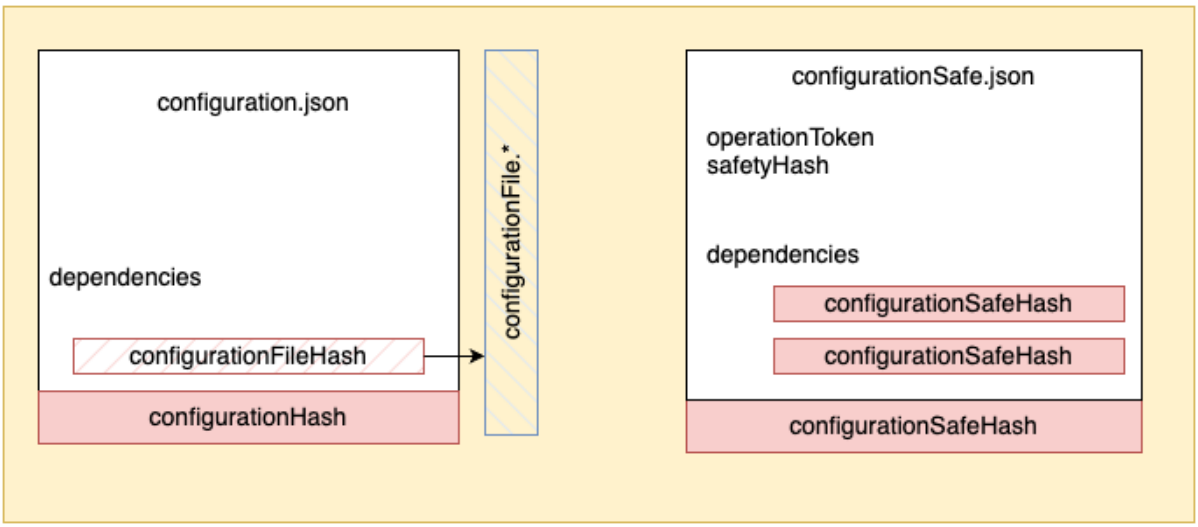
When you take a BuildingBlockConfiguration out of operation, the process begins from the top of the dependency tree and works its way down. The top-most BuildingBlockConfiguration, which depends on other BuildingBlockConfigurations, is taken out of operation first. Then, as you move down the tree, dependent BuildingBlockConfiguration are progressively taken out of operation. This ensures that no dependent BuildingBlockConfiguration is left running when its parent BuildingBlockConfiguration has already been taken out of operation, which could otherwise lead to errors or instability.

SPT2TS-127434 - Activation (Bottom-Up in the Dependency Tree)

When activating a configuration version, the process starts from the bottom of the dependency tree and works its way up. The lowest-level Activation, which are depended upon by other BuildingBlockConfigurations, are activated first. As moving up the tree, the BuildingBlockConfigurations that rely on these lower BuildingBlockConfigurations are then activated. This ensures that when a higher-level BuildingBlockConfiguration is activated, all its dependencies are already activated, providing a stable foundation for the system.

SPT2TS-126687 - Safe BuildingBlockConfigurations: the "configurationSafe.json" documents

In case of a safe BuildingBlockConfiguration (BBC) we need an additional "configurationSafe.json" next to the "configuration.json" document.

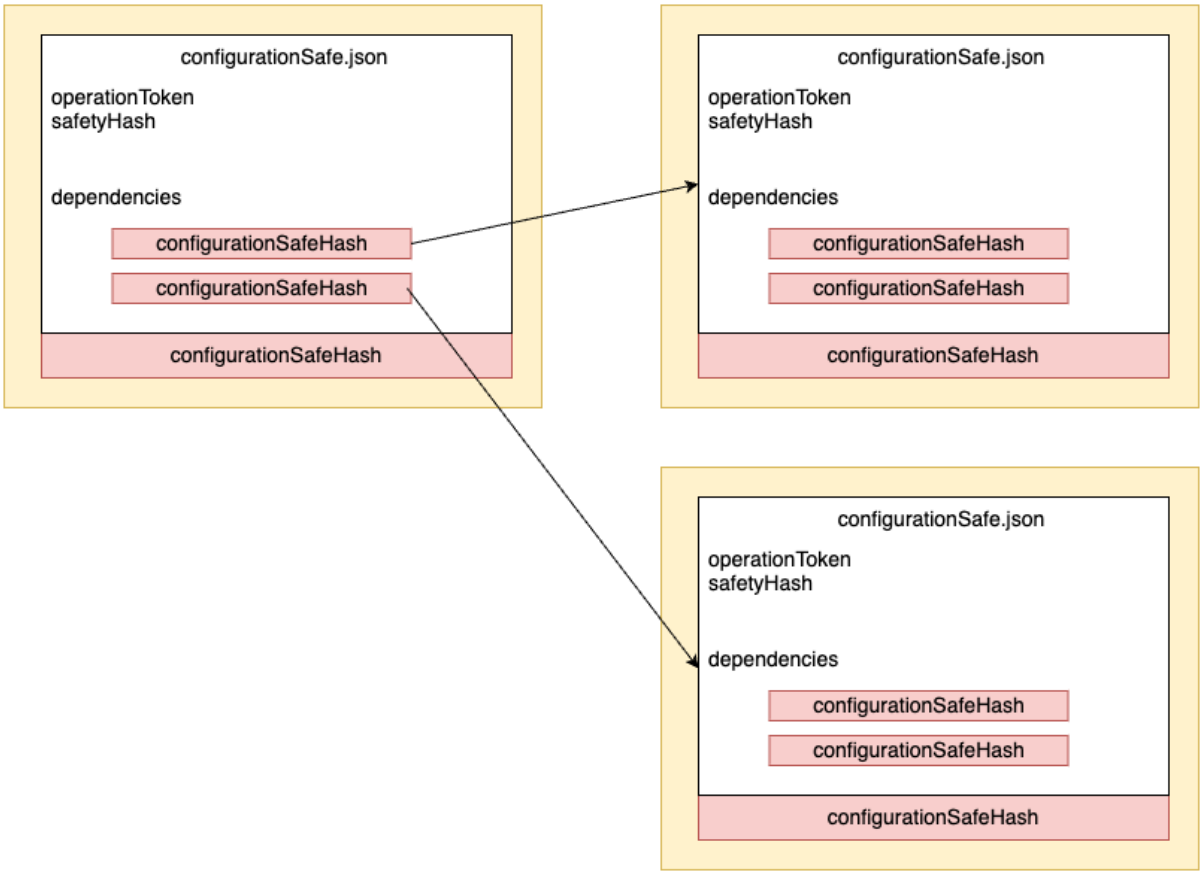


The “configurationSafe.json” file includes attributes necessary for safety attestation, ensuring compliance with the requirement that the correct BBC is installed on the correct BB. The "configurationSafe.json" will be encrypted so that these attributes are not readable for the non-safe configuration management processes. The "configuationSafe.json is not transferred to the BuildingBlock.

The hash of the "configurationSafe.json" of the dependency is contained in within the configurationSafeHash.

For the "configurationSafe.json" this attribute must be added to the dependency on a higher level of the dependency tree to ensure the integrity of the safe dependency tree structure, because it is not possible to change any value in the dependency tree below without also updating the configurationSafeHash and signing on all upper integration levels.

The arrows in the following picture show which items must be included in the calculation of the configurationSafeHash fingerprints to ensure the integrity of these entities.



This recursive (re-)calculation of the configurationSafeHash is a burden to the process that is needed for the safe configuration, because it ensures that all integrators on higher levels check and sign the integration.

As a result the integrators on each of the higher levels of the "safe" dependency tree have to update and approve their "configurationSafe.json" files and the corresponding configurationSafeHash values.
This is needed for the safe contents and imposes a burden on the data preparation process. But it is the only way to express the responsibility of each integrator for the correctness of the composition of the included dependencies.

The "configurationSafe.json" documents also include the operationToken and the safetyHash that are needed during safety attestation (see below).

SPT2TS-127448 - New safe versions must be homologated

New safe versions must be homologated
Updated versions of the safe part of the dependency tree identified by a new top level BuildingBlockConfiguration version must be homlogated.

SPT2TS-127124 - Semantics of the attributes of the BuildingBlockConfiguration "configurationSafe.json" document

These are the descriptions of the attributes that are different or additional to the "configuration.json" document (see above).

Attribute	Description
operationToken	Provided by the supplier or integrator to allow the operation of the BBC
safetyHash	The safetyHash is calculated during data preparation and is used for safety attestation. The safety attestation process is described below.

dependencies	The dependencies additionally contain the configurationSafeHash of the safe dependency.
	
	Here is an example of one entry of a list of dependencies:
	
	"dependencies": [
	{
	"configurationGroupId": "com.integrator-a.area-1",
	"configurationId": "logical.interlocking-1",
	"configurationVersion": "2.2.1",
	"configurationSafetyRelevance" : "SAFE",
	"configurationSafeHash": "sjgskjdfbglskblkjsdfbg"
	},
	{
	"configurationGroupId": "com.integrator-a.area-1",
	"configurationId": "logical.point.subsystem-0815.param",
	"configurationVersion": "1.1.5",
	"configurationSafetyRelevance" : "SAFE",
	"configurationSafeHash": "thfskjdghbglskblkjsdnb"
	}
	],
	
	The dependencies contain the configurationSafeHash for the configurationSafe this configuration is depending on.

Table 4 BuildingBlockConfiguration "configurationSafe.json" description

SPT2TS-127126 - configurationSafe.schema.json

Please find the schema as an attachment in the "Work Item Properties" of this item.

The configurationSafe.json uses the json syntax. A json-schema document is provided to describe the syntax. configurationSafe.json documents must comply to this configurationSafe.schema.json:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "array",
  "items": [
    {
      "description": "This object represents the primary configuration details including
```

```
security and operational parameters for a specific firmware of a product.",
      "type": "object",
      "properties": {
        "@schemaLocation": {
          "type": "string",
          "const": "http://schemas.rail-research.europa.eu/configurationSafe/V0.1"
        },
        "modelVersion": {
          "type": "string"
        },
        "configurationGroupId": {
          "type": "string"
        },
        "configurationId": {
          "type": "string"
        },
        "configurationVersion": {
          "type": "string"
        },
        "configurationSafetyRelevance": {
          "type": "string",
          "enum": ["SAFE", "NONSAFE"]
        },
        "safetyHash": {
          "type": "string"
        },
        "operationToken": {
          "type": "string"
        },
        "name": {
          "type": "string"
        },
        "url": {
          "type": "string",
          "format": "uri"
        },
        "dependencies": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "configurationGroupId": {
                "type": "string"
              },
              "configurationId": {
                "type": "string"
              },
              "configurationVersion": {
                "type": "string"
              },
              "configurationSafeHash": {
                "type": "string"
              },
              "configurationSafetyRelevance": {
                "type": "string",
                "enum": ["SAFE", "NONSAFE"]
              }
            },
            "required": [
              "configurationGroupId",
              "configurationId",
              "configurationVersion",
              "configurationSafeHash",
              "configurationSafetyRelevance"
```

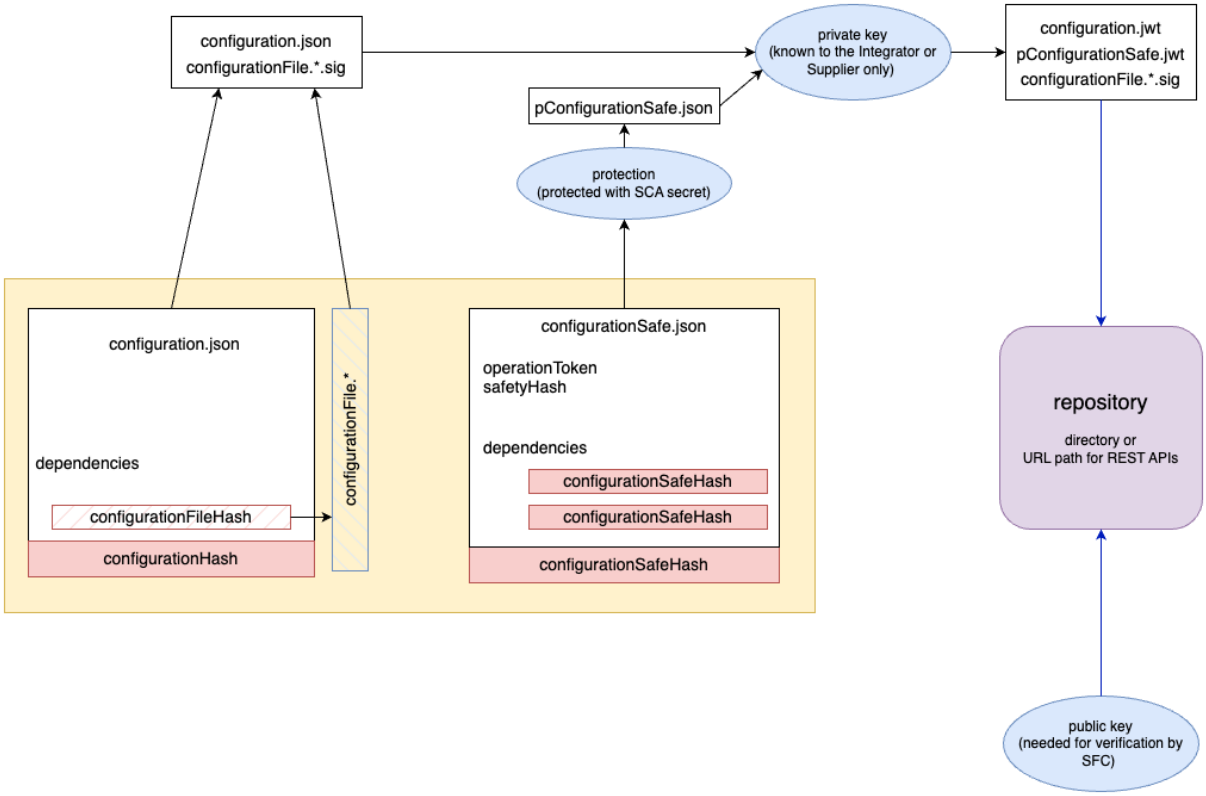
```
            ]
          }
        },
        "repositories": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "name": {
                "type": "string"
              },
              "url": {
                "type": "string",
                "format": "uri"
              },
              "sequenceNr": {
                "type": "integer"
              }
            },
            "required": ["name", "url"]
          }
        }
      },
      "required": [
        "@schemaLocation",
        "modelVersion",
        "configurationGroupId",
        "configurationId",
        "configurationVersion",
        "configurationSafetyRelevance",
        "safetyHash",
        "operationToken",
        "name",
        "url",
        "repositories"]
    },
    {
      "description": "The attribute 'configurationSafeHash' in this object contains a hash over the first object, providing an integrity check to ensure that the primary configuration has not been altered.",
      "type": "object",
      "properties": {
        "@schemaLocation": {
          "type": "string",
          "const": "http://schemas.rail-research.europa.eu/configurationSafeHash/V0.1"
        },
        "configurationSafeHash": {
          "type": "string"
        }
      },
      "required": ["@schemaLocation", "configurationGroupId", "configurationId", "configurationVersion", "configurationSafeHash"]
    }
  ],
  "minItems": 2,
  "maxItems": 2
}
```


SPT2TS-125713 - Digital Signatures: ensuring the the authenticity of configuration documents

Only known, authorised entities shall create, and release configurations. On one hand, a receiving entity must be able to verify the identity of the source entity, to ensure it is from a trusted source. On the other hand, it must be able to check if a configuration has been specifically targeted to the receiving

entity, and was not supposed to be handled by some other entity.
To support the identification and authentication process, all entities involved in the configuration management need unique identifiers. Cryptographic techniques are needed for authenticating the involved systems.

The suppliers and integrators on each of the levels of the dependency trees are end-to-end responsible for their "configuration.json" documents and optional binary configurationFiles.
The responsibility is documented by signing the artifacts cryptographic.
As a result we can ensure the authenticity of all files.
The following picture represents this for a repository of one BuildingBlockConfiguration.



The Supplier or Integrator must sign the configuration.json using his private key of its Manufacturer Config Signing Certificate (MCSC, see SPPRAMSS-7675) and creates a configuration.jwt (JSON Web Signature (JWS)) according to RFC 7515 (see https://www.rfc-editor.org/rfc/rfc7515.html). The configuration.jwt includes signature and content, but the content is no longer readable. For this reason the original configuration.json must also be included into the repository directory.

If available, the Supplier or Integrator must also sign the configurationFile.* of each individual BuildingBlockConfiguration they take responsibility for. The result is a separate configurationFile.*.sig file that must also be copied to the same repository directory.

The signing entity must also copy its public key into the repository directory so that the trusted source can be identified and authentication as well as the integrity check of the contents is possible. The trust of the public key must be verifiable by the Certificate Authority (CA). The CA could be a shared service.

5.2.2 distribution-Job

SPT2TS-125846 - distribution-job.json document

The distribution-job.json file describes the conditions of **WHEN** to distribute BuildingBlockConfiguration items in json-syntax.

In most cases the operator is responsible for a distribution-job.json document.
It exists on the highest level level of the dependency tree.
The distribution-job.json is addressed to and evaluated by an SFC that covers a specific area or CCS on-board.

The distribution-job references the top-level BuildingBlockConfiguration to be distributed.

SPT2TS-125971 - distribution-job configurations

A distribution-job identified by distributionGroupId and distributionVersion must contain one or more distributionIds as target where to distribute the BuildingBlockConfigurations described in the distribution.json document.

SPT2TS-125969 - For one homologated model only

A distribution-job.json file is applicable for one homologated model only.

These may be e.g. vehicles that have the identical composition of component hardware and parametrisation revisions, and that have the same homologation. Infrastructure elements (interlockings) are in most of the cases unique and contain only one configuration.

SPT2TS-125970 - distribution-job activation

The operator is responsible, that the distribution-job is in line with the operational requirements (see above).

The activation may need an additional condition to be met within the activation time window to cover different operational scenarios, e.g.:

- a vehicle is running late and cannot start the activation before arriving in the target station
- A vehicle provided for activation might be needed on short notice
- A train is running late and the interlocking cannot update

The DistributionJob can contain an additional condition that must be met to start the activation: a time window, driver confirmation, GOA4 confirmation. This could be allocated to the DMI and could be specified further in TrainCS.

SPT2TS-127442 - distribution-job repository location

distribution-job documents can be found in a subfolder "distribution-job" in the top level BBC it distributes. In the folder "distribution-job" we can find a subfolder with the @id for each distribution-job distributing the top level BBC. Each distribution-job must be signed by the Operator with its Operator Config Signer Certificate (OCSC). The distribution-job.jwt includes signature and content, but the content is no longer readable. For this reason the original distribution-job.json must also be included into the repository directory.

SPT2TS-125850 - distribution-job.json and distribution-json document semantics

The following table provides an explanation of the attributes of the distribution-job.json documents:

Element	Description
distributionGroupId	The distributionGroupId must be generally unique amongst different organizations and models in case of more than on configuration is distributed in the DistributionJob. Examples: com.integrator-a.vehicle-model-c com.operator-b.area-1 All configurations distributed within one distribution-job are the same. That means that all configurations must be for the same overall model. So the dependency tree contains the same composition of models that need the identical firmware and parametrization. That would be true for ONE interlocking configuration, because interlockings are most often unique. It would also be true for all vehicles complying to one overall MODEL within a fleet of vehicles. In case of a retrofit a new homologation must be created and a new model must be created.

distributionIds	This array contains distributionId that each must be unique together with the distributionGroupId and represent the target for the distribution.
	
	Each distributionId item additionally contains the information when (start and end for preloadTimeWindow and activationTimeWindow) to distribute. All times are given as UNIX Timestamp.
	
	A distributionId can contain a repository, if the configuration preloading and activation should be processed locally e.g. on a vehicle. This might be the case for vehicles that might be parked in an area of poor cell phone reception. In this case it could be useful to partition a repository on the train. A time window is specified when the synchronisation of that repository is made.
	
	Please find an example of such a distributionId item here:
	
	{
	"distributionId": "vehicle-a",
	"preloadTimeWindowStart": "1711978352175",
	"preloadTimeWindowEnd": "1711998352175",
	"activationTimeWindowStart": "1711978352175",
	"activationTimeWindowEnd": "1711998352175",
	"activationStartCondition": "Driver",
	"repository": {
	"syncTimeWindowStart": "1711978352175",
	"syncTimeWindowEnd": "1711998352175",
	"id": "vehicle-a-repo",
	"name": "vehicle a repo",
	"url": "https://vehicle-a.vehicle-model-c.integrator-a.com"
	}
	}

	The repository is identified by its "id" and gets a human readable "name". The "url" defines where to find the repository - for example a REST api service accessible via https (see example above).
@id	A version might be transferred to multiple items that have the same homologation. The @id will make these distribution-Jobs unique. The expected start of the distribution-job can be used, e.g. "@id": "2024-03-31-01-00"
name	Human readable name of the distribution.
configurationGroupId configurationId configurationVersion configurationSafetyRelevance	The distribution.json links to the Top-Level BuildingBlockConfiguration identified by configurationGroupId.configurationId.configurationVersion. For more information refer to the description in the configuration document.

Table 5 BuildingBlockConfiguration distribution-job.json description

SPT2TS-125851 - distribution-job.schema.json

The distribution-job.json documents use the json syntax. A json-schema document is provided to describe the syntax. distribution-job.json documents must comply to this distribution-job.schema.json:

Please find the schema as an attachment in the "Work Item Properties" of this item.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "array",
  "items": [
    {
      "description": "This object represents the distribution job for vehicles.",
      "type": "object",
      "properties": {
        "@schemaLocation": {
          "type": "string",
          "const": "http://schemas.rail-research.europa.eu/distribution-job/V0.1"
        },
        "modelVersion": {
          "type": "string"
        },
        "distributionGroupId": {
          "type": "string"
        },
        "distributionIds": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "distributionId": {
                "type": "string"
              },
              "preloadTimeWindowStart": {
                "type": "string",
                "pattern": "^[0-9]+$"
```

```
        },
        "preloadTimeWindowEnd": {
          "type": "string",
          "pattern": "^[0-9]+$"
        },
        "activationTimeWindowStart": {
          "type": "string",
          "pattern": "^[0-9]+$"
        },
        "activationTimeWindowEnd": {
          "type": "string",
          "pattern": "^[0-9]+$"
        },
        "activationStartCondition": {
          "type": "string"
        },
        "repository": {
          "type": "object",
          "properties": {
            "syncTimeWindowStart": {
              "type": "string",
              "pattern": "^[0-9]+$"
            },
            "syncTimeWindowEnd": {
              "type": "string",
              "pattern": "^[0-9]+$"
            },
            "name": {
              "type": "string"
            },
            "url": {
              "type": "string",
              "format": "uri"
            },
            "sequenceNr": {
              "type": "integer"
            }
          },
          "required": ["syncTimeWindowStart", "syncTimeWindowEnd", "name", "url"]
        }
      },
      "required": [
        "distributionId",
        "preloadTimeWindowStart",
        "preloadTimeWindowEnd",
        "activationTimeWindowStart",
        "activationTimeWindowEnd",
        "activationStartCondition"
      ]
    }
  },
  "@id": {
    "type": "string"
  },

  "configurationGroupId": {
    "type": "string"
  },
  "configurationId": {
    "type": "string"
  },
  "configurationVersion": {
    "type": "string"
  },
```

```
        "configurationSafetyRelevance": {
          "type": "string",
          "enum": ["SAFE", "NONSAFE"]
        }
      },
      "required": [
        "@schemaLocation",
        "modelVersion",
        "distributionGroupId",
        "@id",
        "configurationGroupId",
        "configurationId",
        "configurationVersion",
        "configurationSafetyRelevance"
      ]
    },
    {
      "description": "This object contains the hash for verifying the integrity of the
distribution job.",
      "type": "object",
      "properties": {
        "@schemaLocation": {
          "type": "string",
          "const": "http://schemas.rail-research.europa.eu/distribution-jobHash/V0.1"
        },
        "distribution-jobHash": {
          "type": "string"
        }
      },
      "required": ["@schemaLocation", "distribution-jobHash"]
    }
  ],
  "minItems": 2,
  "maxItems": 2
}
```


SPT2TS-125847 - Configurations are distributed to different repositories

A repository may be accessible as a directory structure or a REST api (Representational State Transfer). Suppliers and Integrators set up their repositories and make these available as SOURCE repositories for the DistributionJob.

SPT2TS-125848 - DistributionJobs must be able to "harvest" different SOURCE repositories

according to the <repository> in distribution-job and configuration.

SPT2TS-125849 - DistributionJobs must also be able to sync to a TARGET repositories

e.g. vehicles could have poor mobile connection at installation time and need a repository onboard containing all configuration data for the model.

SPT2TS-127199 - Example of a distribution-job and distribution for a number of vehicles sharing the identical homologation

Please find the example as an attachment in the "Work Item Properties" of this item.

SPT2TS-127200 - Example of a distribution-job and distribution for a trackside area of control

Please find the example as an attachment in the "Work Item Properties" of this item.

5.2.3 Distribution

SPT2TS-127432 - Distribution - which safe and non-safe configurations are released

Suppliers use the distribution to express safe and non-safe configurations that are tested and released to be working together. Integrators use this information to build the dependency tree. The distribution can also be used to check that only valid combination of configurations are used.

SPT2TS-127433 - The distribution.schema.json

The distribution is build upon the elements explained above.

Please find the schema as an attachment in the "Work Item Properties" of this item.


```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "array",
  "items": [
    {
      "description": "This object represents the distribution details for a vehicle model.",
      "type": "object",
      "properties": {
        "@schemaLocation": {
          "type": "string",
          "const": "http://schemas.rail-research.europa.eu/distribution/V0.1"
        },
        "modelVersion": {
          "type": "string"
        },
        "distributionGroupId": {
          "type": "string"
        },
        "distributionId": {
          "type": "string"
        },
        "distributionVersion": {
          "type": "string"
        },
        "name": {
          "type": "string"
        },
        "url": {
          "type": "string",
          "format": "uri"
        },
        "dependencies": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "configurationGroupId": {
                "type": "string"
              },
              "configurationId": {
                "type": "string"
              },
```

```
            "configurationVersion": {
              "type": "string"
            },
            "configurationSafeHash": {
              "type": "string"
            },
            "configurationSafetyRelevance": {
              "type": "string",
              "enum": ["SAFE", "NONSAFE"]
            }
          },
          "required": [
            "configurationGroupId",
            "configurationId",
            "configurationVersion",
            "configurationSafetyRelevance"
          ]
        }
      },
      "repositories": {
        "type": "array",
        "items": {
          "type": "object",
          "properties": {
            "name": {
              "type": "string"
            },
            "url": {
              "type": "string",
              "format": "uri"
            },
            "sequenceNr": {
              "type": "integer"
            }
          },
          "required": ["name", "url"]
        }
      }
    },
    "required": [
      "@schemaLocation",
      "modelVersion",
      "distributionGroupId",
      "distributionId",
      "distributionVersion",
      "name",
      "url",
      "dependencies",
      "repositories"
    ]
  },
  {
    "description": "This object contains the hash for verifying the integrity of the
distribution.",
    "type": "object",
    "properties": {
      "@schemaLocation": {
        "type": "string",
        "const": "http://schemas.rail-research.europa.eu/distributionHash/V0.1"
      },
      "distributionHash": {
        "type": "string"
      }
    },
```

```
      "required": ["@schemaLocation", "distributionHash"]
    }
  ],
  "minItems": 2,
  "maxItems": 2
}
```

5.2.4 Repository structure

SPT2TS-127203 - The Repository implements the DI 1 interface

SPT2TS-125844 - configurationGroupId.configurationId.configurationVersion form a hierarchy

The configurations shall be made accessible either as a path in a file structure or via https as a REST api using the same structure in the URI than in the path.

Note: this is applicable to all different products independently of the supplier.

SPT2TS-125965 - Updates

When a "configuration.json" or "distribution-job.json" document is updated the version shall be incremented and it is located in another directory or URL-path.

SPT2TS-127128 - Immutability

Released documents shall be kept immutable in the repository.

SPT2TS-125966 - Repository contents for one BuildingBlockConfiguration

A directory or repository path for a configuration must contain the following files:

file	description	specifics
configuration.jwt	signed file according to SPT2TS-125852 to be distributed by the non-safe Service Function Configuration (SFC) to the BuildingBlock.	
pConfigurationSafe.jwt	signed and protected file according to SPT2TS-125852 to be distributed by the non-safe Service Function Configuration (SFC) to the Safe Configuration Authority.	only needed for safe BBC
configurationFile.*	configurationFile.*	**contains** the bbcHash
configurationFile.*.sig	configurationFile signature	
MCSC public key file		

Note: this is applicable to all different products independently of the supplier.

SPT2TS-125967 - Repository contents for one distribution-job

A directory or repository path for a distribution-job must contain the following files:
- distribution-job.json
- distribution-job.jwt (JWS file of the distribution-job.json)
- MCSC public key file

SPT2TS-127175 - Example

Please find the example as "zip" as an attachment in the "Work Item Properties" of this item.

5.3 Configuration Distribution Process

5.3.1 Design Guideline

SPT2TS-124279 - Decoupling of interfaces based on the scope

As general system design guideline, a decoupling between configuration functions and operational functions regarding respective system interfaces is desired. This means that configuration functions are limited to configuration interfaces.

Remark: E.g. the EULYNX approach uses both SMI and SCI for configuration. SCI is used to have a SIL4 function that the configuration versions of a field element and a the interlocking are consistent.

SPT2TS-126927 - Common update mechanism

As general update process design guideline, a common update mechanism for the update of safety-related and non-safety-related data / software is desired. This means that the update mechanism of safety-related and non-safety-related data / software follows the same principles whereby specific (process) extensions (add-ons) for safety-related updates are required.

SPT2TS-126930 - System central component

As general update process design guideline, an update process which is orchestrated and controlled from a system central component is desired. This means that the update mechanism relies on a centralised topology whereby hierarchical deployments should be supported.

Note: that intentionally contrasts distributed concepts like peer-to-peer voted ones.

SPT2TS-126931 - Update processing

As general update process design guideline, an update process with minimum human interaction is desired. This means that the update process is basically initiated and verified by a human with automatic update processing.

SPT2TS-126932 - SIL footprint

As general update process design guideline, an update process with an as small as reasonable SIL footprint is desired, that still fulfills the safety requirements. This means that the non-safe part should assume as many

functions as possible, while the safe part should be reserved exclusively for safety-related functions.

SPT2TS-126928 - Update of non-safety-related data / software

As general update process design guideline, the update process for non-safety-related data / software is embedded in a common update mechanism. This means that this update is basically initiated and verified by a human with automatic update processing based on non-safety system components.

SPT2TS-126929 - Update of safety-related data / software

As general update process design guideline, the update process for safety-related data / software is embedded in a common update mechanism. This means that this update is basically initiated and verified by a human with automatic update processing based on non-safety system components and (in addition) safety-system components.

SPT2TS-124339 - Handling of dependencies during configuration update

As general update process design guideline, the update process shall consider dependencies during configuration update. This means that there might be a sequence of installation whereby some components must be installed first before others can be installed.

Background information: The sequence of configuration is defined by the recursive dependency tree. If a BuildingBlockConfiguration is dependent on another BuildingBlockConfiguration, this BuildingBlockConfiguration must be installed first. The updates must start in the lowest levels (leaves) of the dependency tree. If all dependencies comply the next level up is being updated. This allows to cascade and aggregate the update process with entities each being responsible for a branch of the dependency tree.

Note: The higher levels are not actually updating anything. Their role is to define dependencies. If these dependencies are complying it is ensured that dependency trees of arbitrary size and depth have versions that comply to each other.

Example: A BuildingBlock might have two BuildingBlockConfiguration including (binary) configurationFiles as payloads. The firmware configuration has to be applied first, the parametrization after that. So the firmware must be a dependency in the parametrization configuration.json document.

SPT2TS-125354 - CENELEC compliant BuildingBlock

The supplier shall ensure that the configuration of its BuildingBlock is safe according to the SIL levels assigned to the functions within the system border of the BuildingBlock, if the generic product is configured within the limits defined in the safety case of the supplier according to EN 50716:2023.

SPT2TS-125364 - Responsibilities on different levels of the dependency tree

The <dependencies> are made explicit in the configuration.json of the BuildingBlockConfiguration.
The concept allows to assign responsibilities to different levels of the dependency tree. This allows that the in many cases the competences on the different levels will be in line with the responsibilities.

SPT2TS-125365 - High Level BuildingBlockConfigurations

The concept of the BuildingBlockConfiguration configuration.json document is applicable to arbitrary number of dependency levels.

Examples:

1. *The BuildingBlockConfiguration of a field element may depend on BuildingBlockConfigurations of type firmware and parametrization.*
2. *The BuildingBlockConfiguration of an interlocking may depend on BuildingBlockConfigurations of type field element and TPS.*
3. *The BuildingBlockConfiguration of an area of control may depend on BuildingBlockConfigurations of type TPS, EAL and ATO-TS.*

SPT2TS-125366 - Releasing only BuildingBlockConfiguration payloads that contain "absolute" content

Each BuildingBlockConfiguration must describe the intended configuration absolutely. No relative updates, always complete; dependencies must be explicit.

SPT2TS-125355 - Releasing High level BuildingBlockConfigurations

Each level must describe the intended configuration of all BuildingBlocks it depends on absolutely - all BuildingBlockConfigurations must be contained.

SPT2TS-126691 - Authenticity and Integrity of configurations

Identification and authentication: besides ensuring configuration integrity and confidentiality, the concept shall ensure that only authenticated configurations from identified sources are applied.

In conclusion, a configuration shall only be activated after a positive verification of the authenticity of its source, ascertainment that the source is a trusted SafeConfigurationAuthority (safe functions) or ConfigurationManagementSystem (non safe functions) entity, and that the BuildingBlockConfiguration is targeted to the correct receiving entity.

The technical activation of BuildingBlockConfigurations as per the BuildingBlockConfiguration configuration.json documents shall be handled by each BuildingBlock independently. The internal procedure(s) potentially strongly vary between different BuildingBlock suppliers. Hence, the actual activation mechanisms are not to be standardised.

SPT2TS-125356 - Efficient modularisation of configuration function components

The partitioning of the components in the logical architecture can depend on the availability and reliability of the network.

Example: for vehicles it could be appropriate to partition the ServiceFunctionConfiguration (SFC) incl. the SafeConfigurationAuthority (SCA) add-on on-board. For trackside a more centralised topology might be appropriate.

SPT2TS-125357 - BuildingBlock only operational with intended BuildingBlockConfiguration(s)

It shall be ensured, that a BuildingBlock is only in operation if its version complies as a dependency to the High Level BuildingBlockConfiguration. This implies as well, that all BuildingBlockConfigurations it depends on shall have a specified version.

SPT2TS-125359 - Commit Function of the Safe Configuration Authority (SCA)

The SCA checks if BuildingBlockConfiguration versions comply to the **safe** High Level BuildingBlockConfiguration dependency tree. It will commit safe BuildingBlockConfigurations, if all BuildingBlockConfigurations that are incompatible to the new Top-Level BuildingBlockConfiguration are out of operation.

SPT2TS-125363 - Determine the intended BuildingBlockConfiguration version

By analysing the dependency trees recursively the installation will start bottom up.

SPT2TS-125358 - BuildingBlock requesting commit for going into operation or staying into operation after lease time (silent update)

A BuildingBlock shall request the SafeConfigurationAuthority for approval to go into operation again in the following cases:

- The BuildingBlock is rebooting (e.g. after power loss, BuildingBlock exchanged, etc.)
- The lease time has expired
- At least one BuildingBlockConfiguration of the BuildingBlock has updated to a different version

SPT2TS-126692 - Rollback

The rollback shall use the identical process: an new distribution-job has to be created that references an unchanged older version.

Note: it could be useful to prepare such a distribution-job before the update to reduce the time needed in case the update fails.

Open question: how many versions should be stored in the BuildingBlocks - should at least the last version still be available to reduce the time needed for preloading etc.?

5.3.2 Safety Critical Functions

The following functions of the configuration process are considered to be safety critical:

SPT2TS-126747 - Creation of the safe BuildingBlockConfiguration documents and tree structures

The DataPreparation must create safe consistent dependency trees including all BuildingBlockConfiguration versions starting from a safe High Level BuildingBlockConfiguration and put this in repositories.

SPT2TS-126719 - Analysis of the safe BuildingBlockConfiguration tree

The SafeConfigurationAuthority or DataPreparation must parse the safe dependency tree that it gets from data preparation in repositories and must calculate the intended BuildingBlockConfiguration versions according to the intended High Level BuildingBlockConfiguration ("will").

Note:
It could be useful to partition this function to the SCA so that the SCA is capable of switch from any version to any other version. If DataPreparation has this function then the possible version deltas have to be calculated in advance.

SPT2TS-126720 - Persistent Knowledge about the currently operated safe BuildingBlockConfiguration versions

The SafeConfigurationAuthority must store persistently which BuildingBlockConfiguration versions are in operation on which BuildingBlock.

SPT2TS-126721 - Calculating "Difference" in case of a High Level BuildingBlockConfiguration version switch

The SafeConfigurationAuthority must be able to safely recursively calculate the difference of the currently existing version of the dependency tree vs. intended version for safe configurations. As a result it will know which BuildingBlockConfigurations need to switch to which intended version.

SPT2TS-126722 - Confirm incompatible BuildingBlockConfigurations are out of operation

The SafeConfigurationAuthority shall safely confirm that all BuildingBlocks with incompatible BuildingBlockConfiguration versions are out of operation.
Note: This is a necessary condition for the Top Level BuildingBlockConfiguration version switch that any new version can be confirmed to go into operation.

SPT2TS-126723 - Feedback: safe Top-Level BBC version switch is possible

The SafeConfigurationAuthority must be able to report that the switch to the new version is possible now.

SPT2TS-126724 - Feedback: Status of activations

During the update process the SafeConfigurationAuthority shall monitor and store persistently, which BuildingBlock has activated which BuildingBlockConfigurations, or any exceptions during the activation attempt.

Note: On this basis a decision can be made to go into operation, although some BuildingBlocks are not operating because they could not activate the intended version.

SPT2TS-126725 - Provide operational token after successful activation

The SafeConfigurationAuthority (SCA) shall lookup and provide the operational token to a BuildingBlock in case the activation of the intended BuildingBlockConfiguration version was successful.

SPT2TS-126726 - (Re-)Confirm operational token

The SafeConfigurationAuthority (SCA) shall provide the operational token to a BuildingBlock in case the BuildingBlock has the intended BuildingBlockConfiguration version.

Note: This function is needed during version changes AND in case a failed BuildingBlock is being replaced.

5.3.3 Distribution

SPT2TS-127204 - The Interface "DI 6"

"DI 6" is the interface for the distribution of safe and non-safe BuildingBlockConfigurations. It develops the existing EULYNX SMI interface further.

5.3.3.1 Basic Principles

SPT2TS-126728 - Operational View

In case of a TopLevel BuildingBlockConfiguration version switch all incompatible BuildingBlockConfigurations must be out of operation.

A BuildingBlock can only go into safe operation if all its safe BuildingBlockConfigurations get a confirmation from the SafeConfigurationAuthority.

A BuildingBlock can only stay in safe operation if all its safe BuildingBlockConfigurations have been confirmed by the SafeConfigurationAuthority and the BuildingBlock meets the lease policy valid for each of the safe BuildingBlockConfigurations.

Non-safe BuildingBlockConfigurations can go into operation once a confirmation from the ServiceFunctionConfiguration has been received.

SPT2TS-126746 - Lease Policy

The confirmation from the SafeConfigurationAuthority to the BuildingBlock can hold a lease policy for each of the safe BuildingBlockConfigurations.

The lease policy might hold conditions based on operational status like: onReboot, onReconnect, onMaintenance, inDepot (geofencing).

We may consider to add a temporal lease condition. If the temporal lease condition expires the BuildingBlockConfiguration needs a confirmation from the SCA to stay in operation. This could be useful if we need to take a BuildingBlockConfiguration out of operation, but this cannot be ensured in the process, e.g. we cannot connect to the BuildingBlock. In this case we could wait until the temporal lease condition has expired to be sure that the BuildingBlockConfiguration is out of operation. Then other BuildingBlockConfigurations with a version switch could be confirmed.

Note: The lease policy could be part of the operational token and needs to be prepared by the integrator.

SPT2TS-125987 - Distribution Process

The configuration distribution process is characterised by the distribution of the BuildingBlockConfigurations (DistributionJob) and ensuring the safety attestation, applicable for the safe BuildingBlockConfigurations.

The configuration distribution process is separated into the following stages:

Stage	Description
Preloading	New BuildingBlockConfigurations are loaded to the BuildingBlock. *Note: Background loading without any restrictions is supported.*
Prohibit Operation	Incompatible safe BuildingBlockConfigurations are taken out of operation. *Note: In case of safe BuildingBlockConfigurations the SafeConfigurationAuthority ensures that all incompatible safe BuildingBlockConfigurations are out of operation before a version switch can be processed.*
Activation	Preloaded BuildingBlockConfigurations are installed at the BuildingBlock. *Note: Operation is not allowed without the confirmation of the SFC/SCA.*
Confirmation	BuildingBlockConfigurations get the confirmation to return into operation. *Note: In case of safe BuildingBlockConfigurations the SafeConfigurationAuthority ensures the safety attestation before the BuildingBlockConfiguration gets the confirmation to go into operation*.

The configuration distribution process relies essentially on the following actors:

Actor	Description
User	Human User Responsible to express his "will" by selecting a distribution-job that links the top-level BuildingBlockConfiguration to be distributed.
SFC	Service Function Configuration Service function that realises the non-safety configuration functions; applicable for safe and non-safe BuildingBlockConfigurations. The SFC could be located in the MDM or a vehicle.
SCA	Safe Configuration Authority Safety function that realises the safety critical configuration functions; only applicable for safe BuildingBlockConfigurations. The SCA is a dedicated safe component within the SFC.
BB	Building Block A BuildingBlock can host safe and non-safe BuildingBlockConfigurations (BBC). For the further information refer to chapter Terms and definitions.

5.3.3.2 Safety Attestation

SPT2TS-125975 - Safety Attestation

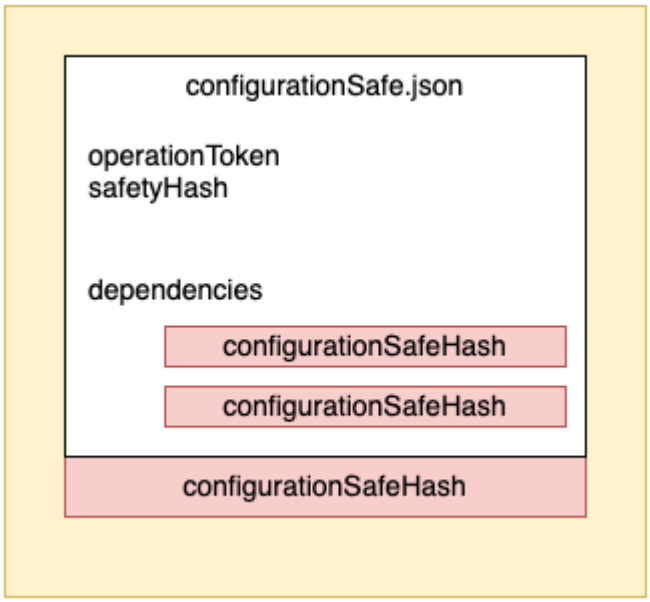
Safety attestation refers to the formal process of verifying and declaring that a safe BuildingBlockConfiguration installed on a BuildingBlock complies with the intended configuration version.

In order to reach this goal, the safety attestation is used to prove safely, that the correct BBC is installed at the intended BuildingBlock and the conditions for activation are fulfilled. During the update of a safe BuildingBlockConfiguration (BBC) on a BB different actors like SFC, SCA and the User are involved in the safety attestation process.

The principles shown here are applicable for safe BuildingBlockConfigurations only.

SPT2TS-125972 - Safe BuildingBlockConfiguration attributes

For safe BuildingBlockConfiguration the attributes “operationToken” and “safetyHash” are composed during data preparation and added to the “configurationSafe.json” document. The document has been put to the repository corresponding to the ID of the BuildingBlockConfiguration (see above for definitions). The content of these attributes have been prepared by the responsible entity (Supplier or Integrator).



SPT2TS-127244 - Data Preparation for a logical component

During data preparation the Supplier delivers the bbcHashPostInstallationExpected, the bbcToken and the configurationGroupId/Id/Version for its generic model.

The Supplier also adds the bbcToken to the binary configurationFile that will be transferred to to BB later.

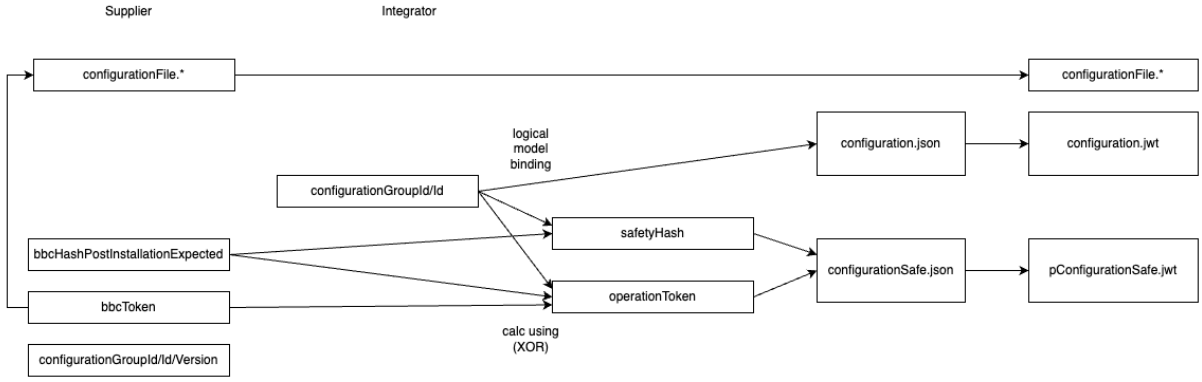
The Integrator creates the logical configurationGroupId/Id/Version for the logical component context he intends to use the generic model.

The binding of the logical component to the generic model is done with calculating:

- the safetyHash from the configurationGroupId and condigurationId of the logical component and the bbcHashPostInstallationExpected.
- the operationToken from the configurationGroupId and condigurationId of the logical component, the bbcHashPostInstallationExpected and the bbcToken

The safetyHash and the operationToken are added to the configurationSafe.json document that will be protected with the bbc secret.

The next picture only shows the flow of the attributes:



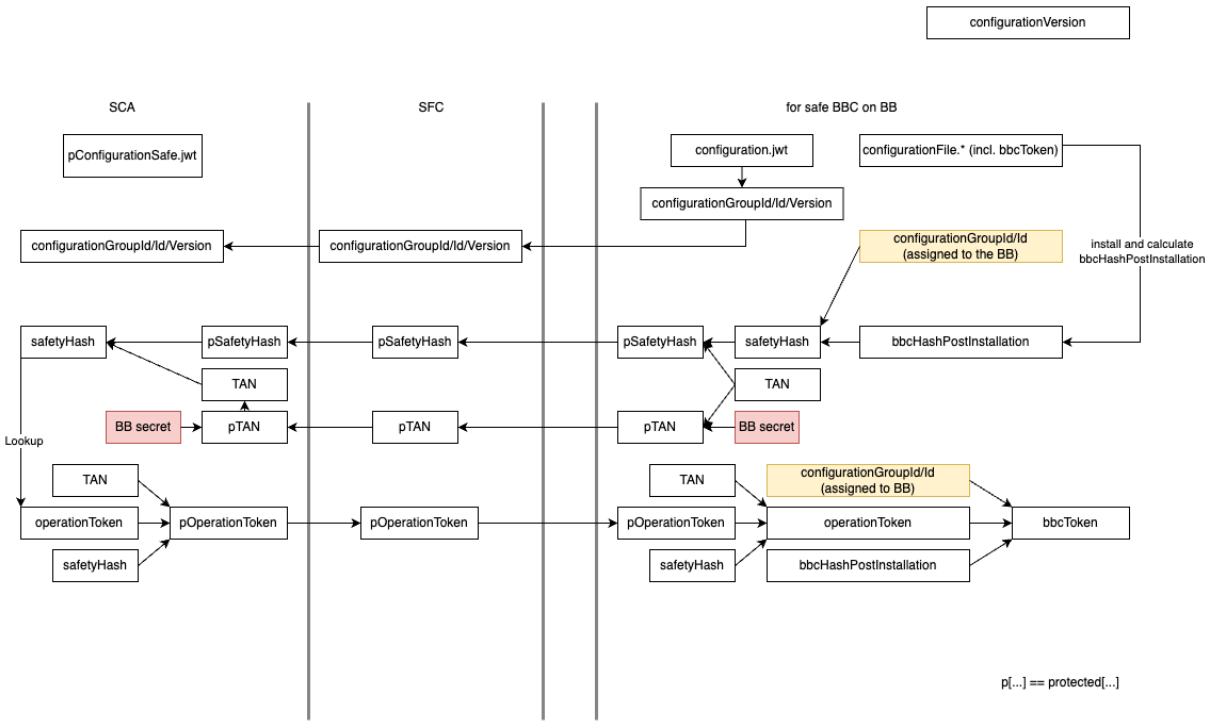
SPT2TS-127245 - Safety attestation during activation and confirmation

Basically the BB repeats the same calculations done in data preparation:

- the safetyHash is calculated from the configurationGroupId and configurationId assigned to the logical component and the bbcHashPostInstallation is calculated on the state of the filesystem after the installation. The safetyHash calculated on the safe computer of the BB will only be identical to the one calculated during data preparation if the bbcHashPostInstallation matches the bbcHashPostInstallationExpected.

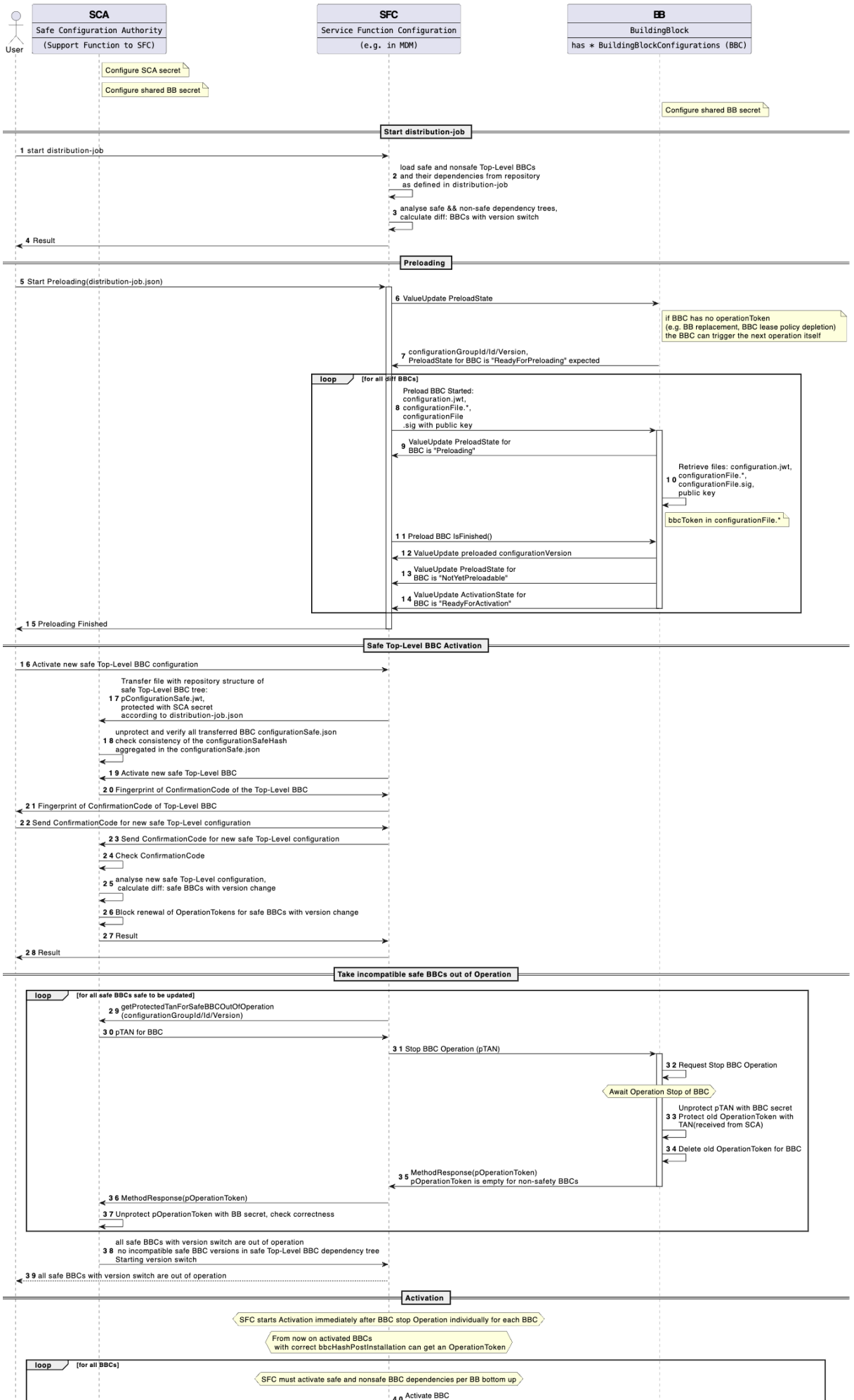
  Note: For example in a EULYNX object controller the configurationGroupId and configurationId assigned to the logical component is stored on the basic data identifier which is plugged in the object controller.
- A TAN, protected with the BB secret, ensures that the BB request to commit is uniquely connected to the SCA commit.
- The SCA checks if the safety hash calculated on and transferred from the BB matches the safety Hash calculated during data preparation. This safetyHash can be found in the protected pConfigurationSafe that can be read by the SCA only.
- If the safety Hash matches the SCA looks up the operationToken from the pConfigurationSafe.
- The SCA bundles the operationToken, the TAN, and the safetyHash into the protected pOperationToken to start the commit.
- Excluding the different values from the pOperationToken the bbcToken is calculated. The bbcToken is contained in the binary configurationFile of the supplier. It will only match if TAN, logical component and bbcHashPostInstallation is correct.

The flow of information is shown in the following picture:



5.3.3.3 Sequence

SPT2TS-127206 - Sequence diagram of the configuration update process


SCA
Safe Configuration Authority
(Support Function to SFC)
SFC
Service Function Configuration
(e.g. in MDM)
BB
BuildingBlock
has * BuildingBlockConfigurations (BBC)
User
Configure SCA secret
Configure shared BB secret
Configure shared BB secret
Start distribution-job
1 start distribution-job
2 load safe and nonsafe Top-Level BBCs and their dependencies from repository as defined in distribution-job
3 analyse safe && non-safe dependency trees, calculate diff: BBCs with version switch
4 Result
Preloading
5 Start Preloading(distribution-job.json)
6 ValueUpdate PreloadState
if BBC has no operationToken (e.g. BB replacement, BBC lease policy depletion) the BBC can trigger the next operation itself
7 configurationGroupId/Id/Version, PreloadState for BBC is "ReadyForPreloading" expected
loop
[for all diff BBCs]
8 Preload BBC Started: configuration.jwt, configurationFile.*, configurationFile .sig with public key
9 ValueUpdate PreloadState for BBC is "Preloading"
10 Retrieve files: configuration.jwt, configurationFile.*, configurationFile.sig, public key
bbcToken in configurationFile.*
11 Preload BBC IsFinished()
12 ValueUpdate preloaded configurationVersion
13 ValueUpdate PreloadState for BBC is "NotYetPreloadable"
14 ValueUpdate ActivationState for BBC is "ReadyForActivation"
15 Preloading Finished
Safe Top-Level BBC Activation
16 Activate new safe Top-Level BBC configuration
17 Transfer file with repository structure of safe Top-Level BBC tree: pConfigurationSafe.jwt, protected with SCA secret according to distribution-job.json
18 unprotect and verify all transferred BBC configurationSafe.json check consistency of the configurationSafeHash aggregated in the configurationSafe.json
19 Activate new safe Top-Level BBC
20 Fingerprint of ConfirmationCode of the Top-Level BBC
21 Fingerprint of ConfirmationCode of Top-Level BBC
22 Send ConfirmationCode for new safe Top-Level configuration
23 Send ConfirmationCode for new safe Top-Level configuration
24 Check ConfirmationCode
25 analyse new safe Top-Level configuration, calculate diff: safe BBCs with version change
26 Block renewal of OperationTokens for safe BBCs with version change
27 Result
28 Result
Take incompatible safe BBCs out of Operation
loop
[for all safe BBCs safe to be updated]
29 getProtectedTanForSafeBBCOutOfOperation (configurationGroupId/Id/Version)
30 pTAN for BBC
31 Stop BBC Operation (pTAN)
32 Request Stop BBC Operation
Await Operation Stop of BBC
33 Unprotect pTAN with BBC secret Protect old OperationToken with TAN(received from SCA)
34 Delete old OperationToken for BBC
35 MethodResponse(pOperationToken) pOperationToken is empty for non-safety BBCs
36 MethodResponse(pOperationToken)
37 Unprotect pOperationToken with BB secret, check correctness
38 all safe BBCs with version switch are out of operation no incompatible safe BBC versions in safe Top-Level BBC dependency tree Starting version switch
39 all safe BBCs with version switch are out of operation
Activation
SFC starts Activation immediately after BBC stop Operation individually for each BBC
From now on activated BBCs with correct bbcHashPostInstallation can get an OperationToken
loop
[for all BBCs]
SFC must activate safe and nonsafe BBC dependencies per BB bottom up
40 Activate BBC

SPT2TS-126984 - Description of the sequence diagram

Pre-Conditions:
- BBC artifacts have been prepared and are accessible in the BBC repository via interface DI 1.
- Distribution-Job is prepared and accessible in the repository via interface DI 1.
- SCA: Secret for safe BuildingBlockConfigurations (BBC) provisioned in SCA.
- BuildingBlock (BB): Secret for safe BuildingBlockConfigurations provisioned in BuildingBlock (BB) safe.

Start distribution-job
1. Start distribution-job. The distribution-job document contains a Top-Level BBC as dependency.
2. Load Top-Level BBC and its dependencies from repository as defined in distribution-job.
3. The SFC analyses the dependency tree and calculates which BBCs have a version switch (difference).
4. the Result is shown to the user: which safe and non-safe BBCs need an update.

Preloading
5. Start preloading.
6. Get ValueUpdate PreloadState.
7. PreloadState for BBC is "ReadyForPreloading" expected
-- start the preloading of BBCs according to the diffs calculated in step 3 .
8. Start BBC preloading: configuration.jwt, configurationFile.*, configurationFile.sig, public key, for safe BBC add pConfigurationSafe.jwt.
9. PreloadState is "Preloading" now.
10. The files configuration.jwt, configurationFile.*, configurationFile.sig, public key and for safe BBC pConfigurationSafe.jwt are transferred.
11. PreloadState is finished now.
12. Value Update preloaded configurationVersion.
13. PreloadState is "NotYetPreloadable".
14. ActivationState is "ReadyForActivation".
15. Finish Preloading.

Safe Top-Level BBC Activation
16. The User chooses a safe Top-Level BBC corresponding to a preloaded distribution-job.
17. The SFC transfers the complete safe Top-Level BBC dependency tree with all pConfigurationSafe.jwt tree to the SCA.
18. The SCA checks the configurationSafe hashes, that are aggregated bottom up in the dependency tree.
19. The SFC requests the activation of the new safe Top-Level BBC from the SCA.
20. The SCA creates a Fingerprint of the ConfirmationCode (can only be decoded by the User).
21. The SFC shows the Fingerprint of the ConfirmationCode to the User.
22. If the User decides to proceed the User sends the ConfirmationCode for the new Top-Level configuration to the SFC.
23. The ConfirmationCode is sent to the SCA.
24. The SCA checks if the ConfirmationCode is valid.
25. SCA analyses the dependency tree and calculates, which BBCs need to update the configurationVersion.
26. The SCA knows which safe Top-Level-BBC is currently activated. Once the new safe Top-Level-BBC is under

activation, the SCA stops giving OperationTokens to BBCs, until all the dependencies of the new safe Top-Level-BBC are completely activated.
Knowing the currently installed safe Top-Level-BBC and the new safe Top-Level-BBC, the SCA is able to calculate which BBC needs to be updated, and thus the list of BBCs for which new bbcHashPostInstallation have to be provided.
27. The SCA reports its status to the SFC.
28. The SFC reports the SCA status to the User.

Take incompatible safe BBCs out of Operation
29. The SFC requests a pTAN for a safe BBC identified by configurationGroupId/Id/Version to request taking out of operation from the SCA.
30. The SCA generates a TAN that is protected with the SCA secret shared with the safe BBs and responds with a protected TAN (pTAN).
31. Request to stop operation of the BBC that needs to be updated, pTAN is a parameter (SFC is used as a "grey channel") .
32. Request to stop operation of the BBC that needs to be updated.
- BBC stops operation considering operational aspects.
33. BBC internal request processed: unprotect pTAN using secret (shared with SCA), protect old operationToken with TAN and secret (shared with SCA) -> pOperationToken.
34. BBC Delete OperationToken for safe BBCs.
35. MethodResponse (grey channel).
36. MethodResponse: success/fail stop operation.
37. Unprotect pOperationToken using secret (shared with BB) and TAN, check correctness.
38. All safe BBCs with version switch are out of operation: no incompatible safe BBC versions in safe Top-Level BBC dependency tree: now Starting version switch.
39. SFC reports to user.

Activation
40. Activate BBC starts the installation of the corresponding BBC.
41. The ActivateState for the BBC is set to "Activation".
42. The BBC is installed. How the installation is done exactly is product specific.
43. A TAN is chosen randomly. The TAN is used to protect the safety hash against reply attacks, e.g. due to erroneous caching. The TAN needs to be sufficiently long to be unique (no collisions with previous TANs). The TAN is protected with a preconfigured secret. In case of a safe BBC the preconfigured secret is only known by the SCA, in case of a non-safe BBC the secret is known by the SFC.
44. In case of a safe BBC the bbcHashPostInstallation of the BBC is calculated by the BB.
45. In case of a non-safe BBC a hash after installation of the BBC is calculated by the BB.
46. The new BBC configurationVersion is set.
The hashes have to be unique for this BBC and they need to be reproducible. The hashes are used to prove that the installation of the BBC has been successfully done. The exact calculation is product specific, as it depends on the installation process. It must be pre-calculated.
47. Set ActivationState for this item to "NotYetActivatable", the activation is done.
48. Set ConfirmationState for this item to "ReadyForConfirmation", the confirmation starts.
49. SFC requests values for BBC: identifier (configurationGroupId/Id/Version), pSafetyHash and pTAN.
50. Read the response (grey channel in case of safe BBCs).
51. If non-safe BBC: SFC checks the non-safe bbcHashPostInstallation (if required).
52. If safe BBC: Provide identifier (configurationGroupId/Id/Version), bbcHashPostInstallation and pTAN to the SCA.

53. SCA unprotects and checks the safe bbcHashPostInstallation vs. the bbcHashPostInstallationExpected (that has been calculated before in data preparation for the BBC and can be found into the dependency tree rooting in the Top-Level-BBC).
54. Result of checking the bbcHashPostInstallation against the bbcHashPostInstallationExpected.

BBC Confirmation
55. Request the new OperationTokens for all BBC that have been updated identified by BBC(configurationGroupId/Id/Version).
56. For each safe BBC: Calculate the protected operationToken (pOperationToken) by using the TAN, the operationToken and the bbcHashPostInstallation. - use (safe) secret, include TAN received in step 51.
57. Reply with the pOperationToken safe.
58. Provide the pOperationToken to the BBC via ConfirmBBCActivation - grey channel.
59. The BBC starts the confirmation process by setting BBC ConfirmationState to "Confirming".
60. The OperationToken is calculated from the pOperationToken by using the TAN that has been generated in step 42. The OperationToken is BBC specific.
61. Calculate the bbcToken from the operationToken by using configurationVersion and the bbcHashPostInstallation.
62. Check the bbcToken. The bbcToken have been transferred before in the configurationFile.* and is known to the entity responsible for the BBC only. If the bbcToken check fails, the operationToken (used for calculating the bbcToken) was meant for a different BB or BB is correct, but the BBC does not match.
63. If "configurationGroupId.configurationId" check was successful, the OperationToken is stored on the BBC, so it can be checked whenever the BBC operation is enabled (e.g. during startup).
-- If during startup no BBC OperationToken is known to the BBC, it can request it from the SCA (through the SFC). If the installed BBC is correct, according to the current Top-Level-BBC, the SCA will reply with the OperationToken (through the SFC), if the installed BBC is incorrect, the BBC has to be updated first before going into operation.
64. The ConfirmationState for the item is set to "NotYetConfigurable", the confirmation is done.
65. SFC reports to user.

6 Open Items

7 Tables

Please update the table of figures.

Please update the table of figures.