Guidelines for Model-Driven Development of Safety IO Drivers at Kongsberg Maritime

SIMULA RESEARCH LABORATORY
AND
KONGSBERG MARITIME AS

December 2010

Authored by: Lionel Briand, Håkon Spiten Mathisen, Anne-Heidi Evensen Mills, Shiva Nejati, Øivind Rui, Mehrdad

KONGSBERG

[simula . research laboratory] - *by thinking constantly about it*

1. Introduction

The maritime sector is becoming increasingly more reliant on software to improve development productivity, enable more sophisticated operations, and provide flexibility in handling evolving needs. The sector now employs advanced Integrated Software-Dependent Systems (ISDSs) to control fire and gas detection, emergency shutdown, and vessel propulsion, positioning and navigation. With the rapid evolution from mechanical to computer-based operations in the maritime sector, there is now an increasing need for a rigorous engineering discipline that can reduce the costs and at the same time improve the dependability of maritime software.

This report presents a study conducted jointly by Kongsberg Maritime (KM) and the Simula Research Laboratory (SRL) as part of a larger and ongoing investigation on how to improve the software engineering practices in the maritime sector. The broad aim of the investigation is to identify the challenges that complicate the development of maritime software, and devise practical and innovative solutions based on *Model-Driven Engineering* to address these challenges.

Model-Driven Engineering (MDE) is a powerful paradigm, where models are used as the primary artifacts of development. MDE is already in wide use in other engineering fields such as electrical and mechanical engineering as well as in software-intensive industries such as the aerospace and automotive industries. When a pragmatic approach is taken, MDE leads to dramatic improvements in the productivity and dependability of software systems by providing more precise descriptions of the requirements, specifications, designs, and test cases.

Realizing the full potential of MDE requires a cost-effective methodology tailored to the needs of the underlying domain, and appropriate tools that support this methodology. For the maritime domain, a complete methodology has to account for the needs of three distinct but interacting stakeholders: the *suppliers*, the *integrators*, and the *certifiers* of maritime ISDSs. The suppliers are in charge of constructing ISDSs or components thereof; the integrators put together supply chains of components or ISDSs delivered by different suppliers; and the certifiers ensure that the relevant dependability (primarily safety) standards have been met by the suppliers and integrators, and that the resulting systems are acceptably safe to operate.

The collaboration between KM and SRL concerns the *supplier's perspective* and specifically on how MDE can put KM in a better position to achieve on time and on budget delivery of more dependable ISDS components and systems.

1.1. Context of Collaboration

The current KM-SRL collaboration is in the context of KM's platform for Safety Instrumented Systems, called KSafe. KSafe itself is built on KM's Control System Application platform, called AIM. KSafe applications are computerized systems designed specifically for safety monitoring and automatic corrective actions on unacceptable hazardous situations. These systems include, among others, emergency and process shutdown, and fire and gas detection systems.

KSafe provides several software components that can be configured to support the needs of each specific system. Two major groups of components in KSafe are *modules* and *drivers*. The modules implement the general functions for safety automation systems. Various modules exist for different types of input and output elements (analogue and digital), cause and effect tables, voting, monitoring, etc. Modules communicate with actual devices (e.g., motor starters, sensors, valves, pumps) through drivers.

Within the context of KSafe, our goal is to study how MDE can facilitate the following:

1. ***Certification,*** so that KSafe applications (assembled from modules and drivers) can be certified faster, with a higher degree of trust, and at lower costs.

2. ***Impact Analysis,*** so that after each change to the requirements or design, we can automatically analyze what development artifacts are impacted and assess the consequences on safety. Impact analysis may lead to re-testing and re-verifications of the impacted components or revising their specifications.

3. ***Communication and Training,*** so that developers across different KM divisions or physical sites can communicate with one another more effectively, and further to conduct the training of new personnel in a faster and more structured way.

4. ***Verification,*** so that such tasks as test suite generation, test management, and inspections can be done more systematically and be assisted by automated tools.

1.2. Scope of this Report

Providing end-to-end modelling guidelines for KSafe will necessarily involve studying both KSafe modules and drivers. To this end, we are conducting two studies, focused respectively on how to best model the architecture, structure and behaviour of modules and drivers. Both studies place special emphasis on ***safety certification***, particularly the need for traceability from safety-relevant requirements to design. Such traceability is crucial for systematically arguing about the satisfaction of safety-relevant requirements and has been recognized as a priority area for our investigation.

This report concentrates on KM drivers and develops modelling guidelines for capturing the design of these drivers and relating the design to the (safety-relevant) requirements. Our study of KSafe modules is ongoing and will be addressed separately in a future report.

We illustrate our methodology using one of KM's drivers, called the *Safety Fire Central* (SFC) driver. The structure and behaviour of SFC is representative of the significant majority of the drivers developed in-house at KM. As the result of our study, a complete set of design models with traceability to requirements have been developed for SFC. These models can be readily adapted to a variety of other drivers at KM, and can be further used as a basis for defining a product family for drivers, with the commonalities and variabilities between the drivers explicitly specified.

1.3. Structure of the Report

The remainder of the report is structured as follows:

We begin in Section 2 with background information on KSafe and the KSafe driver in our study. Section 3 briefly introduces SysML – the language we use for modelling, and describes why SysML is particularly suitable for describing ISDSs. In Section 4, we provide a methodology for model-driven design of KSafe drivers and establishing traceability from requirements to design. In Section 5, we reflect on the modelling experience gained through the study and our observations. We conclude the report in Section 6 with a summary and a highlight of our next steps.

2. Background on KSafe

In this section, we provide an overview of KSafe and describe in some detail one of the KSafe IO drivers, named SFC. This driver will be used throughout the report for illustration.

2.1. Overview

KSafe includes a large collection of software modules implementing the various units of behaviour required in safety automation applications. These modules are defined as software algorithms with pre-programmed (but customizable) functions, and contain terminals for connections to other modules and to physical IO devices via IO drivers. Figure 1 shows two typical module configurations in KSafe. The configuration on top is used for fire and gas detection systems and the one on the bottom – for emergency and process shutdown systems.

Generally speaking, a KSafe application works as follows: it receives a set of digital or analogue signals from IO units (e.g. fire centrals and field sensors). These IO units are distributed across the areas under control and can be installed at both non-hazardous

and hazardous locations. Since the IO units are most commonly outside the central control room, they are called Remote IO (RIO) units. Note that in certain cases (e.g. when third-party hardware is involved) RIO units may not be physically present, but it is still necessary to have RIO drivers for proper communication in the AIM framework.

Signals received from the RIOs are transmitted to the controller part, made up of chains of software modules. The first module in a chain is in charge of checking that the input is not faulty. These modules can be further configured to suppress the effects of an alarm on the measured input value. In Figure 1, the first modules in the control chain are *sd_ainput*, *sd_dmeas*, and *autro_pt*: *sd_ainput* handles generic analog input; *sd_dmeas* handles generic digital input; and *autro_pt* handles (digital) detector points connected to a particular type of device, called the AutroSafe Fire Central. Modules that deal with analog input include a discretizer that translates the input data to digital values based on specified thresholds.

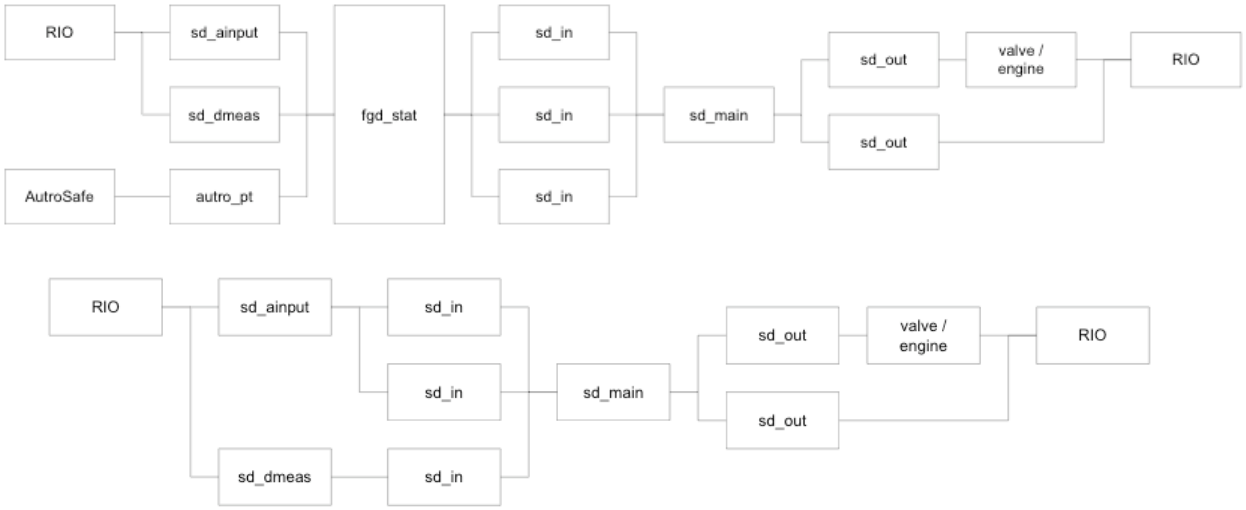

FIGURE 1: TWO TYPICAL CONFIGURATIONS OF KSAFE APPLICATIONS.

In the next stage, a voting module may be applied (e.g., *fgd_stat* in the configuration shown on top of Figure 1) to find out how many of the input detector modules have raised alarm(s). Voting is mainly used for confirmation of input sensor values.

The output from the voting module (or from the input detector modules, when no voting is required) is sent to a generic function module, called *sd_in*, that represents one input to a cause & effect matrix. The actual matrix containing the relationships between causes and effects is kept in separate module, called *sd_main*. Each *sd_in* is programmed to write to certain cells in the matrix. Upon presentation of one or more causes as input, *sd_main* determines the actions (effects) to take place in response the causes. Each effect is represented using a generic module, called *sd_out*. The output from *sd_out* is then communicated to the RIO controlling the actuator that must render the effect. If the actuator is a complex device like a complex valve or an engine,

additional processing may be necessary before the effects are communicated to the RIO. Based on the type of the valve or engine being used, this further processing can be performed using various (valve and engine) modules. These modules are still considered as special kinds of output modules.

IO drivers connect software modules to hardware and mechanical devices. The drivers usually appear at the beginning and at the end of module chains, connecting the RIO to the input/output modules, but other configurations of drivers are also possible where a module in the middle part of the module chain interacts with a driver.

KSafe drivers typically interact with devices not directly but rather through a common lower-level driver, named *ComAs*. ComAs defines the interface between the (Remote) Control Unit, called RCU, on which KSafe components execute and the physical transmission media such as serial lines or Ethernet. The virtual communication ports and channels defined by ComAs can be referred to by all the drivers on an RCU. The example driver we discuss in Section 2.2 and use for exemplification of our proposed methodology in the subsequent sections interacts with a fieldbus driver, named ModBus, which in turn communicates with the ComAs driver. Many KSafe drivers have direct interactions with ComAs as well.

2.2. The Safety Fire Central (SFC) Driver

The SFC IO driver is a KSafe driver that enables the transferring of specific commands from an RCU to a fire detection panel called the Thorn Fire Central[1]. The panel can be used in a variety of applications, e.g., general cargo and passenger vessels and offshore installations, and can be set up to control various types of fire detection sensors.

The SFC driver handles a particular interaction with Thorn Fire Central, where a KSafe application requests the suppression of input signals from certain fire detectors. This is useful when one or more detectors attached to a Thorn Fire Central are in an abnormal state, for example when the detectors are malfunctioning or are under maintenance (hot work), or before the KSafe application is set into operation.

The SFC driver is a unidirectional driver in the sense that it only relays commands to the Thorn Fire Central, without reading input (or alarms) from it. Obtaining input from Thorn Fire Central detectors is handled by separate drivers that are not discussed in this report.

In Figure 2, we show a high-level view on the working of the SFC driver: the request for suppression of a particular detector is communicated to the driver by the input module for that detector. Most often, this input module is an instance of *sd_ainput* or *sd_dmeas*. The majority of KSafe software modules, including all the input modules,

[1] The technical brand name for the device is the Thorn MX/T2000 Panel.

have an outgoing signal called `InhibitOut` to indicate if the (digital or analog) measurement terminal should be suppressed. The SFC driver periodically scans the `InhibitOut` signals from the input modules. If any changes in signals are detected since last scan, the driver relays the changes to Thorn Fire Central which will in turn apply the inhibit actions.

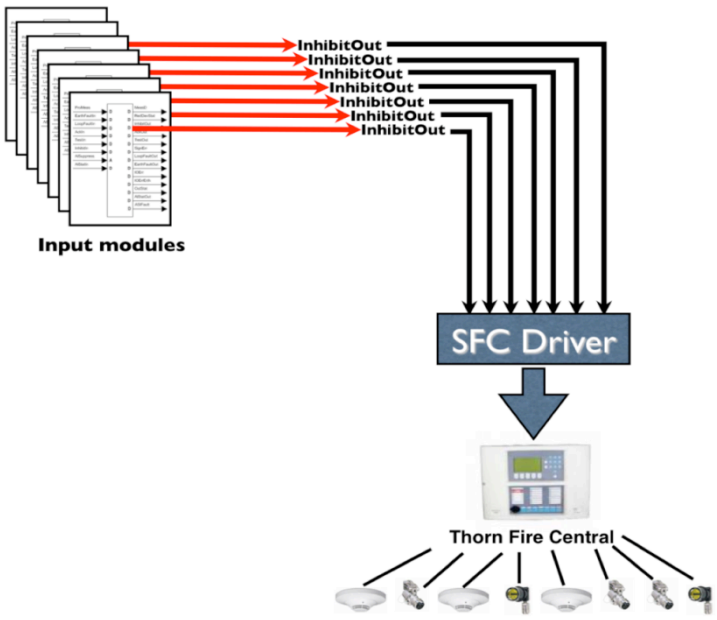

FIGURE 2: THE SAFETY FIRE CENTRAL IO DRIVER

In a KSafe application, the `InhibitOut` signals from the input modules can be controlled in three ways: (1) Using a certain terminal, named `InhibitIn`, of the input module; (2) using a configurable module parameter, named `Inhibit`; and (3) from the module operation menu of the KSafe application. The mechanism through which an `InhibitOut` signal is turned on or off does not affect the operation of the SFC driver and hence not elaborated further here.

In Section 4, we will use the SFC driver for illustrating the methodological steps that we propose for modelling of KSafe drivers.

3. Background on SysML

We use the System Modelling Language (SysML) [1] as the basis for the methodology described later in Section 4. In the section that follows, we provide a brief introduction to SysML, focusing on the SysML diagrams utilized in our proposed methodology.

SysML is a general graphical language for systems engineering that can be used for specification, analysis, design and verification of complex systems containing hardware, software, personnel, facilities and procedures [1]. SysML was standardized in 2006 by the Object Management Group (OMG) – the most widely accepted consortium on standardization of modelling languages. SysML extensively reuses UML 2, while also providing certain extensions to it. This is illustrated in Figure 3.

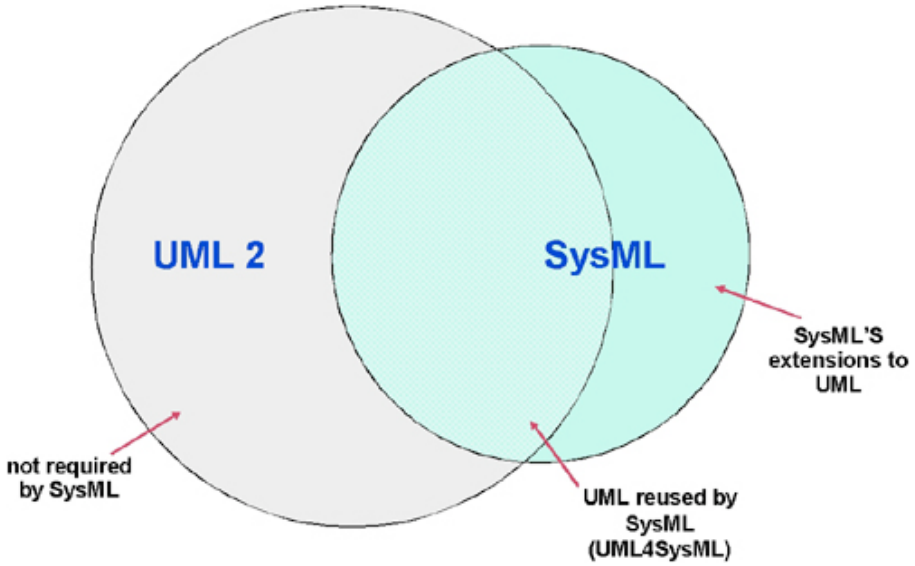

FIGURE 3: SYSML REUSES UML 2 WITH AN ADDITIONAL EXTENSIONS [1]

The relationship between SysML diagram types and UML 2 is shown in more detail in Figure 4. As shown by the figure, there are two types of SysML diagrams that do not exist in UML 2. These are (1) the Requirement Diagram, where we can capture/develop the requirements and relate them to other requirements or model elements, and (2) the Parametric Diagram, where we can capture constraints for property values. Activity Diagrams, Internal Block Diagrams, and Block Definition Diagram are modifications of existing diagrams in UML 2. Sequence Diagrams, State Machine Diagrams, Use Case Diagrams, and Package Diagrams in SysML are identical to their UML 2 counterparts.

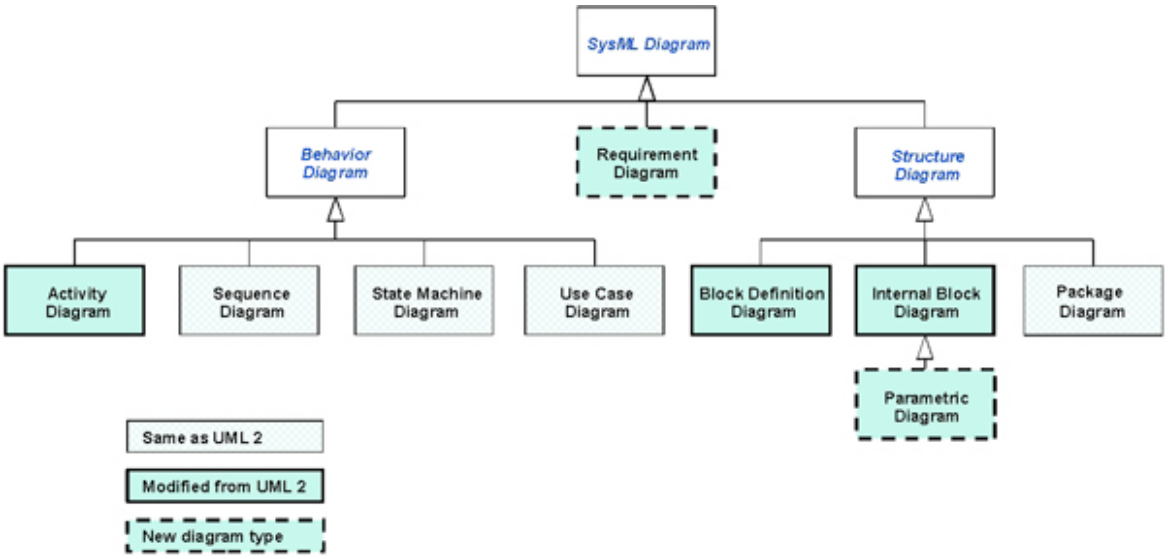

FIGURE 4: SYSML DIAGRAMS AND THEIR RELATIONSHIP WITH UML DIAGRAMS [1]

Compared to UML, SysML offers the following advantages for specifying Integrated Software-Dependent Systems (ISDSs) and systems-of-systems [2] :

- SysML expresses systems engineering semantics (interpretations of modelling constructs) better than UML, thereby reducing the bias UML has towards software.
- SysML removes many software-centric constructs. As a result, SysML is smaller than UML in terms of the number of diagram types (9 vs. 13) and total constructs. This also makes SysML easier to learn.
- SysML provide various means, not available in UML, for enabling automated Verification and Validation (V&V).
- SysML model management constructs support the specification of models, views, and viewpoints and are architecturally aligned with IEEE-Std-1471-2000 (IEEE Recommended Practice for Architectural Description of Software-Intensive Systems).

Our methodology in Section 4 utilizes four SysML diagrams, namely Block Definition Diagrams (BDDs), Internal Block Diagrams (IBDs), Requirement Diagrams (RDs) and Activity Diagrams (ADs). In the remainder of this section, we provide a short introduction to these four diagrams. Additional details about the diagrams will be provided where needed in Section 4 as we go though the methodology and show example diagrams for the SFC driver.

3.1. Block Definition Diagram (BDD)

In SysML, there is a concept called block, which is a *"modular unit of system description"* [2]. Blocks are used to describe structural concepts in a system and its environment. Blocks can represent both physical and logical elements, including mechanical parts, hardware, software, humans, domain concepts, development artifacts (e.g., documents and source code) and so on. A block can contain structural and behavioural features, which are described as properties and operations. The block and its features together with its relationships with other blocks are defined in a *Block Definition Diagram (BDD)*.

In Section 4, BDDs are used for describing the context and the structure of each driver as well as the decomposition of the driver's activities. The diagrams in Figure 6, Figure 8, Figure 9, and Figure 14 are all examples of BDDs.

3.2. Internal Block Diagram (IBD)

The internal structure of a composite block, i.e. a block that contains other blocks, is described using an *Internal Block Diagram (IBD)*. IBDs are particularly useful for modelling the connections and interactions between the parts of a composite block and the usage of the parts. This is done by describing (1) the flow of items (e.g. physical items, data or information) and/or (2) the interfaces for provided and required services. There may be direct connections between the parts using a connector, which will show that the parts are connected, but not describe how. For physical parts, this can mean that they are attached physically.

In Section 4, IBDs are used for describing the architectural connections of a driver. The diagram in Figure 7 is an example of an IBD.

3.3. Requirement Diagram (RD)

One of the new concepts in SysML is the Requirement block, used to define conditions or functions that the system must satisfy. Each requirement block contains at least an id property and a textual description property. Requirements are shown visually in a *Requirement Diagram (RD)*. When establishing links between requirements and design elements, requirements can be shown in other diagrams, or similarly, other diagrams (or parts thereof) can be shown in Requirements Diagrams. Some of the common relationships used for linking requirements to one another and to design elements are shown in Table 1.

TABLE 1: REQUIREMENT RELATIONSHIPS

Relationship	Explanation
Containment	Used to break a requirement into simpler requirements. All the contained requirements should not add or remove any meaning to the original requirement.
Derive	Used to depict that a requirement is derived from another requirement. Useful for mapping the assumptions made about the system based on the requirements.
Trace	Used to describe a general-purpose relationship, often used to relate a requirement to external documents.
Satisfy	Used to link a model element to a requirement to show that it satisfies the requirement.
Verify	Used to link a test case to a requirement to prove that a model element satisfies it.
Refine	Used to depict that a model element is refined from a requirement. Useful to map assumptions made to reduce ambiguity in a requirement.

In Section 4, RDs are used for describing traceability between the requirements of each driver and the relevant fragments of the design. The diagrams in Figure 16, Figure 17, and Figure 18 are examples of RDs.

3.4. Activity Diagram (AD)

Activity Diagrams (ADs) describe the behaviour of a system or component by showing workflows of activities, performed sequentially or in parallel. Activities can be decomposed into smaller activities. Atomic activities are called actions. The decomposition of activities is sometimes shown using a BDD (for example, see). An activity can have a set of parameter nodes attached to it to specify its inputs and outputs. In SysML, activity diagrams can express the flow of not only information but also physical elements (e.g., fuel) and energy (e.g., pressure). SysML activity diagrams further provide support for modelling continuous activities.

In Section 4, RDs are used for describing the traceability links between the requirements of each driver and the relevant fragments of the design. The diagrams in Figure 16, Figure 17, and Figure 18 are examples of RDs.

4. SysML-Based Methodology for Modelling Drivers

SysML is only a modelling notation and does not provide a methodology on *how* to model a system (or system-to-be). To be able to effectively apply SysML (or any other modelling notation, for that matter), one needs a specific methodology tailored to the problem domain and the objectives to be achieved from modelling.

In this section, we describe such a methodology, based on SysML, for modelling KSafe drivers. The main objective pursued from modelling is to facilitate safety certification by

providing precise and unambiguous specifications of the drivers' design (architecture, structure, and behaviour) and establishing traceability links to the requirements. These links provide the evidence one needs for arguing that the requirements are properly addressed by the design.

Figure 5 shows an overview of our proposed methodology for model-driven development of KSafe drivers. We assume that the requirements for the driver under development have been already specified and provided as input. If the requirements specification cannot be built beforehand (e.g. due to time constraints), one may still apply the methodology in Figure 5 by first defining a small set of core functions to be realized by the driver, and then iteratively refining these functions into a coherent set of requirements during design.

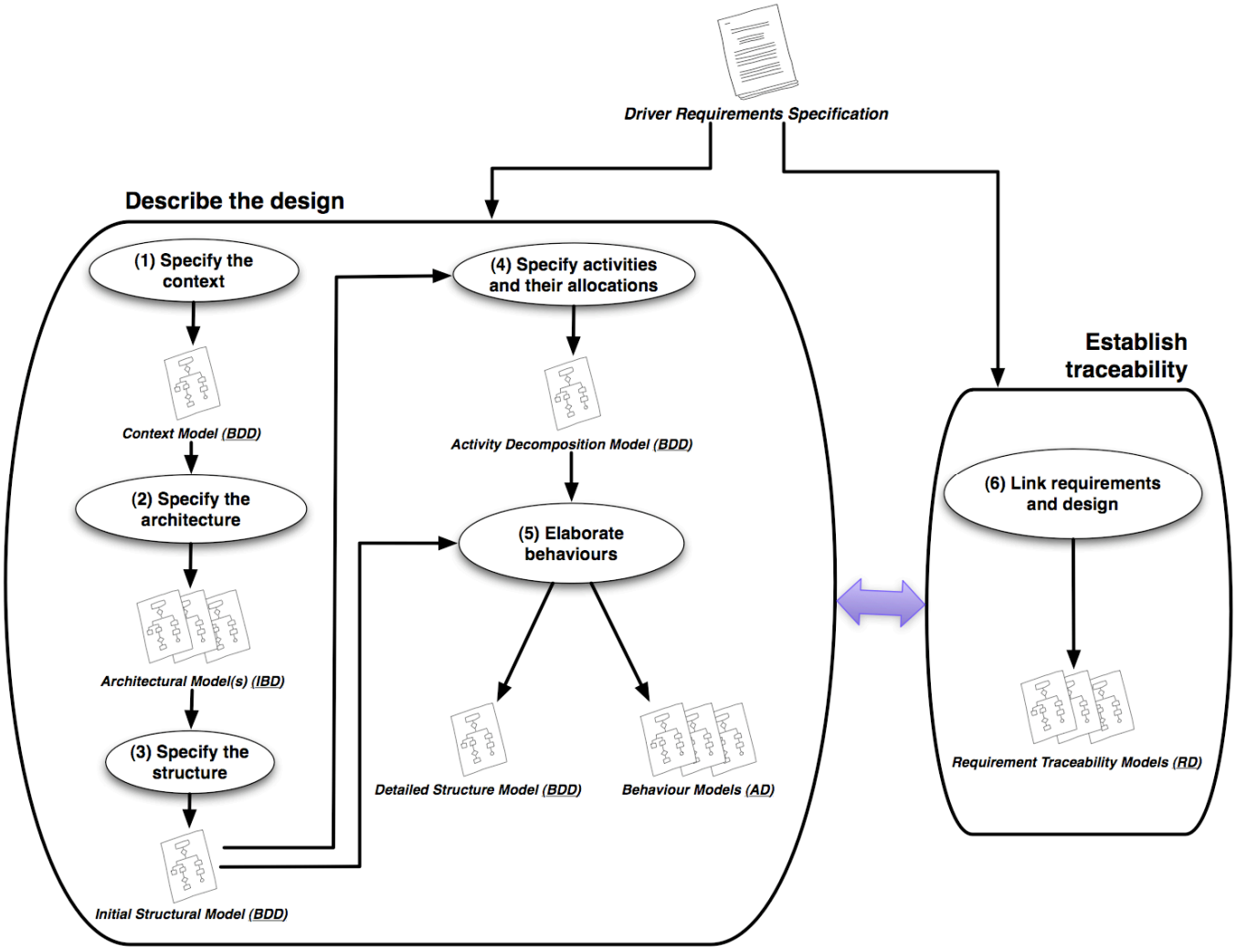

FIGURE 5: METHODOLOGY OVERVIEW

A second issue to note about the requirements is that, since KSafe applications are dedicated safety monitoring and control systems, the significant majority of the requirements for KSafe drivers (and similarly KSafe modules) are *safety-relevant*. That is to say, these requirements in some way contribute to the satisfaction of the overall system safety goals. From a safety certification standpoint, developers can elect to leave out from their analysis the requirements that are not relevant to safety. In other words, the driver requirements specification provided as input to the process in Figure 5 can be restricted to safety-relevant requirements. However, if the inclusion of non-safety requirements poses little overhead (which we believe is usually the case for drivers), it is preferable to apply the process to the whole set of requirements. This ensures that a complete design of the driver is developed (as opposed to just the safety-relevant design aspects), and is highly beneficial for maintenance, communication and testing of the driver.

Our methodology is composed of two parallel but inter-related tasks. The high-level task on the left of Figure 5 ("Describe the design") is concerned with the construction of design models, and the one on the right ("Establish traceability") is concerned with the creation of traceability links between the requirements and design.

The design is carried out in five steps. These steps are depicted as being conducted sequentially in the diagram of Figure 5, but it is important to note that in reality, the discoveries made at later stages of the development may affect the decisions made in earlier later stages. As a result, the SysML diagrams developed in the process will co-evolve and none will be considered final until the design is complete.

The design steps are interleaved with the traceability step (numbered 6 in Figure 5). If the driver being modelled is sufficiently small and the modeling activities span only a few days, the modeler may choose to establish the traceability links *after* the design is complete. However, for a complex driver with a longer development life cycle, it is recommended that the traceability links be created *during* design. Specifically, once a design fragment relevant to a particular requirement is completed, the traceability between the fragment and the requirement should be modelled.

In the remainder of this section, we describe each of the 6 steps in the methodology of Figure 5. We illustrate each step using examples from the SFC driver and provide general guidelines about how to carry out these steps for other KSafe drivers.

4.1. Specify the Driver's Context

The first step in the modelling of a KSafe driver is defining its *context* using a context diagram. This context diagram shows the main system blocks that are related to the driver but are external to it. In SysML, contexts are expressed using BDDs (see Section 3.1).

Figure 6 shows the context diagram for the SFC driver. As seen from the figure, the `KSafeDomain` is defined as being composed of a collection of KSafe modules and a collection of KSafe drivers. These composition relationships are expressed using aggregation links (solid lines with solid diamonds at the container end). The abstract types for the modules are provided through five blocks specializing the `Module` block. We already explained these module types in Section 2.1.

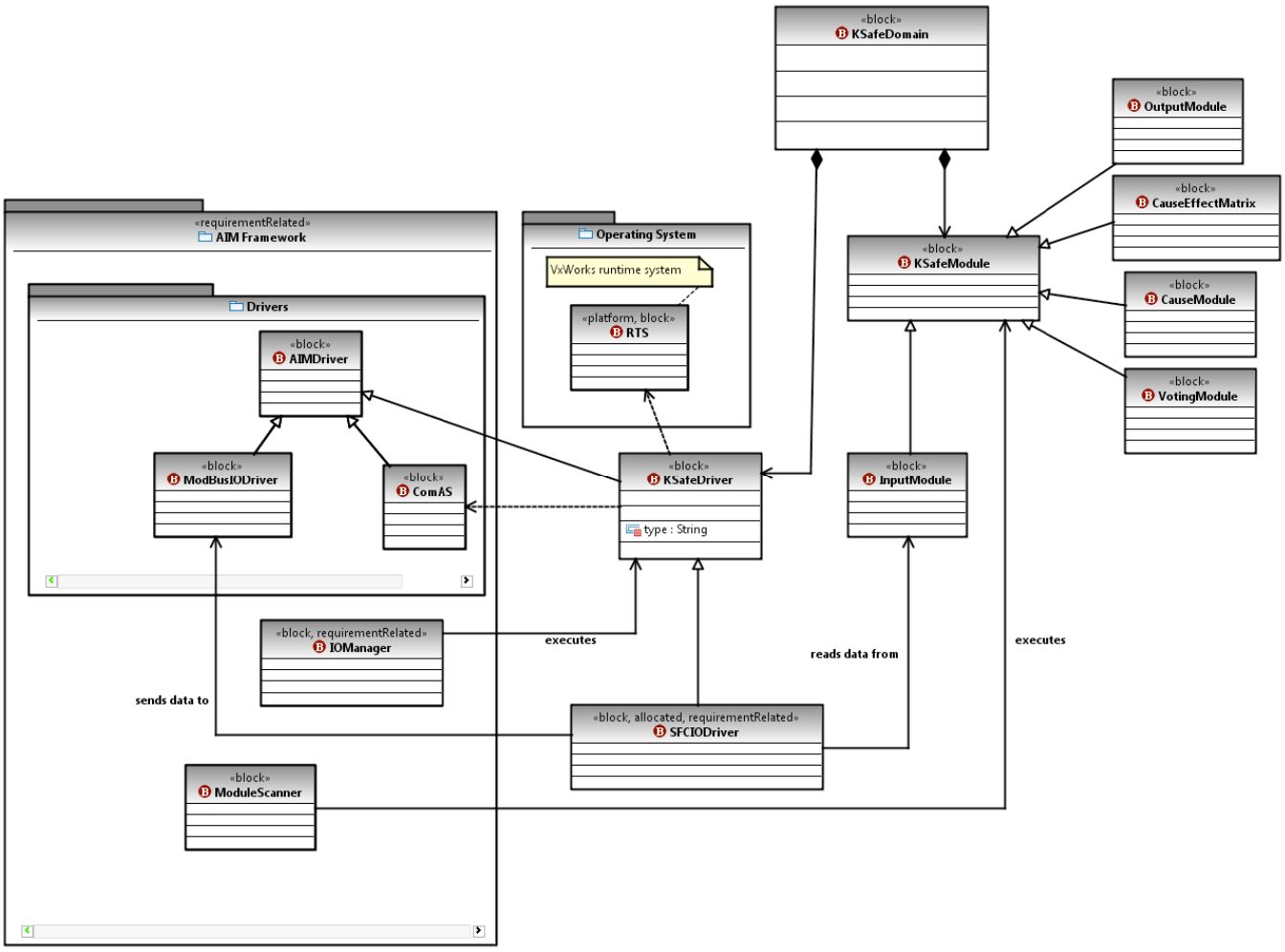

FIGURE 6: CONTEXT FOR THE SFC DRIVER DESCRIBED AS A SYSML BDD

The `SFCIODriver` is a specialization of the `KSafeDriver` block. It receives input from input modules such as sd_ainput and sd_dmeas. This relationship is modelled using an association link (solid line) between the `SFCIODriver` and `InputModule` blocks. The communication between `SFCIODriver` and `ModBusIODriver` (see Section 2.1) is conceptualized using an association link as well. The diagram also captures the fact that both the modules and the drivers are executed within the AIM framework. The execution of the former type of components is done through the `ModuleScanner` block and the execution of the latter – through the `IOManager` block.

To implement its function, KSafe drivers rely on facilities provided by the Run Time System (RTS) of the operating system, and the ComAS driver (see Section 2.1). This is modelled using dependency links (dashed lines) from `KSafeDriver` to the corresponding blocks.

Guidelines on Describing the Context for KSafe Drivers

The substantial part of the context diagram in Figure 6 is shared amongst all the drivers in the KSafe domain and can thus be reused without change. What needs to be revisited for every driver are the following:

- The module(s) and driver(s)[2] that communicate with the driver in question. For `SFCIODriver`, the modules were the `InputModule`s and the driver was the `ModbusIODriver`, but these could be different for other KSafe drivers. Further, `SFCIODriver` can communicate with a "class" of modules (i.e., the input modules) as opposed to just specific ones. If `SFCIODriver` could only communicate with specific modules, say sd_dmeas, then the context diagram for `SFCIODriver` would have required an explicit block for that particular module.

- The dependencies between the driver under development and other blocks. In the case of `SFCIODriver`, only some primitive facilities in the operating system and the AIM framework were used[3]. Other drivers could have additional dependencies that are not shared by all drivers. This could require the inclusion of more blocks from the AIM framework, the operating system, and third-party components, and then adding appropriate dependency links.

Once the context diagram is complete, we can move on to step (2) of the process shown in Figure 5, where we elaborate the communication links between the driver and other blocks and describe the architectural connections of the driver. This is described in Section 4.2.

4.2. Specify the Driver's Architecture

The goal of this step is to specify the dynamic communications between the driver under development and other system blocks. More specifically, we want to refine the conceptual relationships that we defined between the driver and other system blocks in the context diagram of Figure 6 into a set of architectural connectors with specific communication interfaces. Making the interfaces between different system blocks

[2] These other drivers could be KSafe drivers, drivers from the AIM framework, or third-party drivers.

[3] Note that the `SFCIODriver` dependencies to the AIM framework are common dependencies that apply to all KSafe drivers. Hence the dependency links were made from the `KSafeDriver` block.

explicit and free from ambiguity is a major concern in safety-critical systems to ensure that systems components can be integrated properly.

We use SysML IBDs (Section 3.2) for expressing the architectural connections of KSafe drivers. Figure 7 shows the IBD developed for the SFC driver. The central component in the diagram is an instance of the `SFCIODriver` block (named `sfc`).

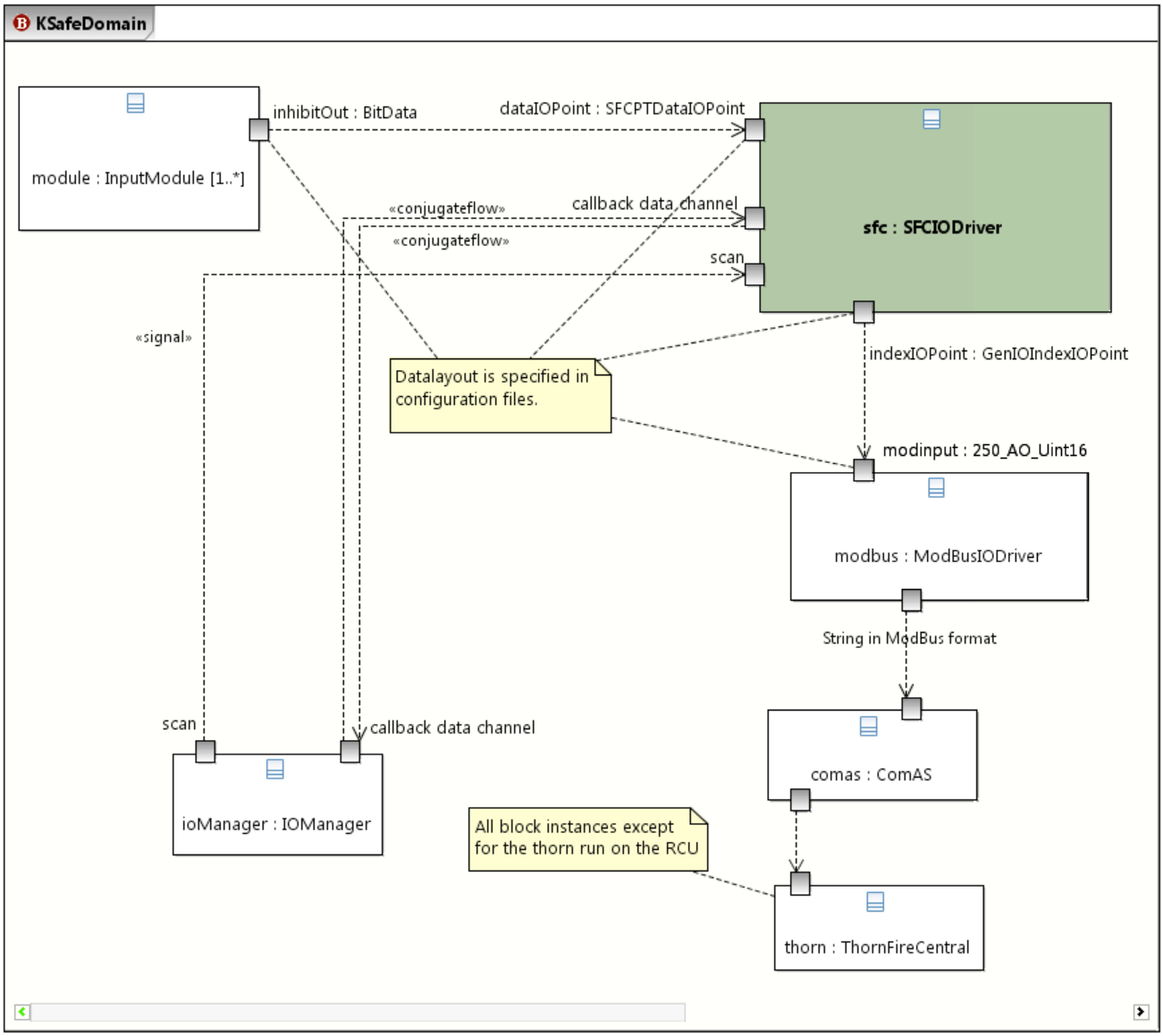

FIGURE 7: ARCHITECTURAL CONNECTIONS OF THE SFC DRIVER DESCRIBED IN A SYSML IBD

In the IBD, we express several important points about the communication between `sfc` and instances of other blocks, particularly:

- What are the ports for communication?

- What data and signals are being communicated over the ports?
- What is the direction of communication?
- How many instances of each block are participating in the communication?
- How many ports (of a certain type) does each block instance have?

Additional information about any of the above points can be expressed by adding stereotypes to the appropriate ports and links or by using comment, as we will illustrate throughout this section.

The `SFCIODriver` communicates with three types of blocks: `IOManager`, `InputModule`, and `ModBusIODriver`. We use multiplicity constraints to describe how many instances of a given block are involved in the communication. The default multiplicity is 1 and is left implicit. In Figure 7, there is exactly one instance of each block involved except for `InputModule`, which has at least one but possibly more instances. This is denoted by the `[1..*]` multiplicity constraint.

Inter-block communications are modelled using ports. The `SFCIODriver` exposes four types of ports as shown in Figure 7:

- `SFCDataIOPoint` defines an AIM-compliant input port for reading the `inhibitOut` value from an input module. The port at the sending end of the communication is a bit value and defined as being of type `BitData`. The direction of the arrow from `module` (instance of `InputModule`) to `sfc` indicates that the communication link is unidirectional. The number of `SFCDataIOPoint` ports that an individual instance of `SFCIODriver` has depends on how many modules are connected to the driver instance. Hence, there is a multiplicity constraint of `[1..*]` assigned to the `SFCDataIOPoint` port type. The modelling tool that we use in this study does not visually show the multiplicity constraints for port types, but provides means to record this important piece of information.

- `GeIOIndexIOPoint` defines a unidirectional AIM-compliant port for forwarding the modules' inhibit data onto the `ModBusIODriver`. The port at the receiving end of the communication is a ModBus-specific point type (represented as a two-byte unsigned integer), called `250_AO_Uint16`. The number of `GeIOIndexIOPoint` ports exposed by the driver is the same as that for `SFCPTDataIOPoint` and hence the same multiplicity constraint applies.

- `callback data channel` is an abstract bidirectional port capturing the non-signal data that is communicated between `ioManager` (instance of `IOManager`) and `sfc`. In KSafe, the communications are managed through callback methods. To indicate that `callback data channel` combines several information flows, we apply a (user-defined) stereotype «`conjugateflow`». The `callback data channel` port only handles non-signal data communications, e.g. reading and writing of driver parameters, and last sent and received inhibit data values. System control signals (such as the `scan` signal described below) need to be modelled individually

and separately. This is because of the important role that these signals play in synchronizing and orchestrating the components of a control system. System control signals are frequently referenced in the behavioural design of the components and hence need to be explicitly modelled.

- `scan` is a periodic clock signal that is fundamental to the working of KSafe applications (both modules and drivers). For drivers, the `scan` signal comes from the `IOManager` and is received by all driver instances running on an RCU, including instances of the `SFCIODriver`. All system control signals are labelled by the (user-defined) «`signal`» stereotype to clearly differentiate them from regular data communications.

The exact point layout for some of the ports in the diagram of Figure 7 are specified by the configuration files of the AIM framework. Describing these layouts is out of the scope of our models, but it is important to specify which ports have corresponding data layouts. We do this by attaching comments to the ports, similar to what we have done in Figure 7. The name of the configuration files could be specified in the comments as well.

The diagram in Figure 7 includes instances of two blocks, `ComAS` and `ThornFireCentral`, which do not communicate directly with the `SFCIODriver`. We show these blocks in order to provide a complete picture of the communication that the SFC driver is meant to enable (see Figure 2). Showing such an end-to-end communication path is beneficial for better understanding of the communication context; but elaborating the architectural links for these additional blocks is unnecessary, as these are considered out of scope for the SFC driver.

Lastly, we need to note that an IBD captures one specific communication architecture, not all the possible architectures in which a driver can be deployed. For example, it is possible and also common for a driver to have several different point layouts to communicate with different components in different deployments. In such cases, each communication architecture is expressed using an individual IBD.

For example, a driver may be configurable to provide both serial and Ethernet connections, and to communicate with different blocks in these two different modes of operation. A good model of such a driver would then need to have two IBDs, one for each mode of operation. This is why the methodology in Figure 5 envisages the construction of multiple IBDs.

Guidelines on Architectural Elaboration for KSafe Drivers

Based on what we said above, the process for modelling the architectural connections of a KSafe driver can be summarized as follows:

1. Determine if the driver has multiple alternative usages and communicates with different sets of blocks depending on the usage. If this is the case, one IBD must be created per usage. The remaining steps are given for a single usage (and hence a single IBD). Generalization to multiple IBDs is straight-forward.

2. Add to the IBD the blocks that directly communicate with the driver. These blocks are readily identifiable from the driver's context diagram (the context diagram for SFC was given in Figure 6). Except for the `KSafeDomain` block that is non-physical, any block that has an association link to the driver block in the context diagram is included in the IBD. Specify the number of participating instances of each of these block using multiplicity constraints.

3. Refine the association links incident to the driver block in the context diagram into information flows in the IBD. Specify the directionality of each information flow.
   - **Note**: It is possible (and likely) for an association link in the driver's context diagram to give rise to several flows in the IBD.

4. For each flow, specify the communication ports on both ends. Distinguish system signals from regular information flows using the «`signal`» stereotype.

5. The datatype and multiplicity for each port must be defined. If the port uses a custom data structure, the data structure must be specified as a block (e.g., the `SFCPTDataIOPoint` and `GenIOIndexIOPoint` datatypes in SFC). These blocks will be included in the driver's structural diagram (see Section 4.3).
   - **Note:** The callback data channel for drivers is provided by the AIM framework and is standard across drivers. Hence, the callback channel is exempt from the port elaboration in step 5. Instead the stereotype «`conjugateflow`» is applied to the callback data flow to indicate that this flow joins together a number of simpler flows for specific callback methods.

Once an architectural view(s) on the operation of the driver is developed, we can move on to the next step, explained in Section 4.3, where we specify the basic internal structure of the driver.

4.3. Describe the Driver's Internal Structure

In the previous steps, we defined some of the structural design elements of the driver. In particular, we defined a block representing the driver itself and also blocks for the driver's ports with complex datatypes. In this step, we extend the structural design of the driver with controller sub-blocks that manage the drivers' ports. The result is an initial structural blueprint for the driver, described as a SysML BDD. Further details will be added to this BDD when we elaborate the driver's behaviour in Section 4.5.

The operations of all KSafe components (modules and drivers) are synchronized by the global `scan` signals delivered to them by the AIM framework. The input and output ports of these components are refreshed periodically in response to the `scan` signals (not necessarily at every single scan). The driver can initiate an out-of-scan refresh in between the scan signals as well, if the driver's parameters change and an immediate refresh becomes necessary.

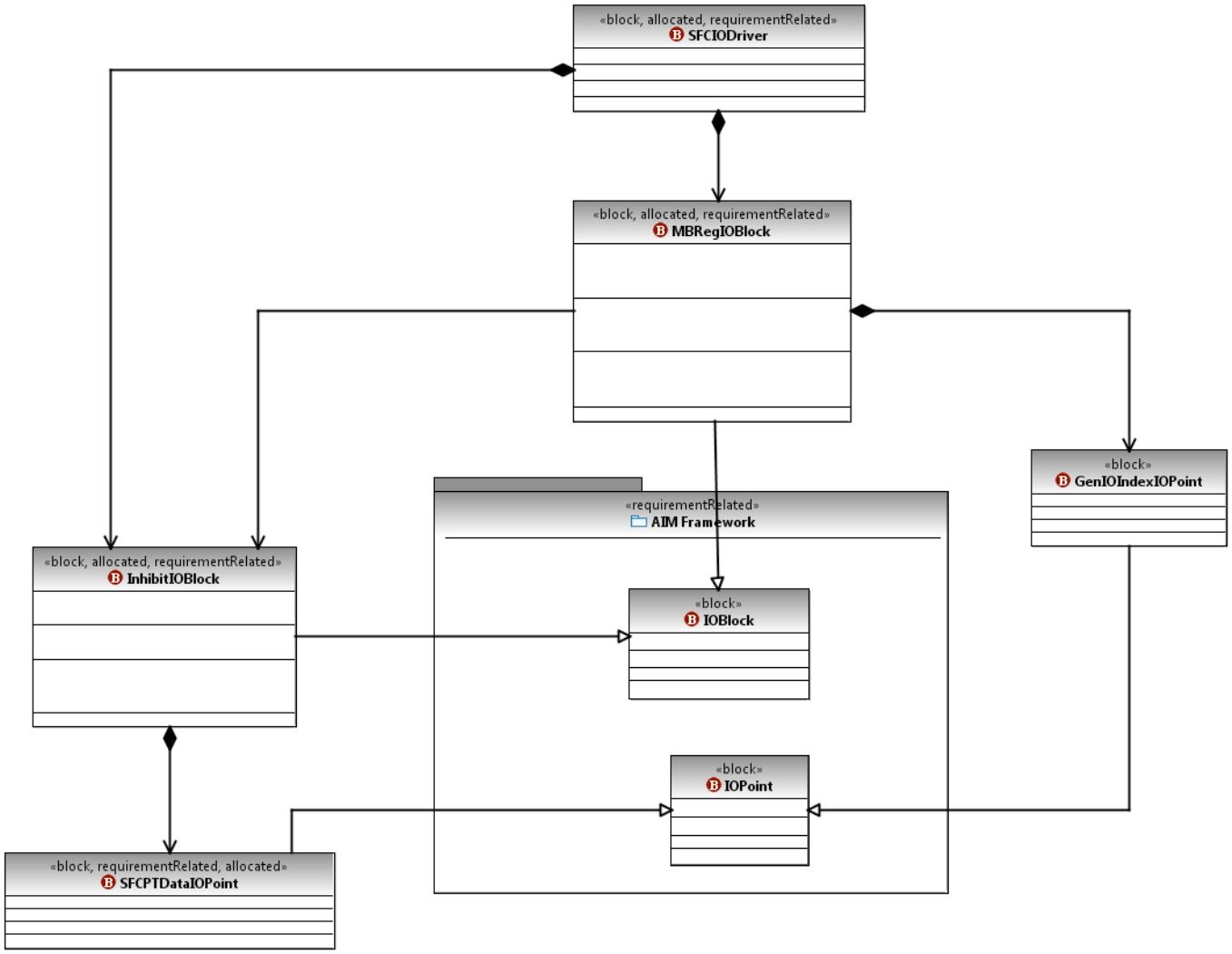

FIGURE 8: DESCRIBING SFC'S INTERNAL STRUCTURE USING A BDD

For drivers, the behaviour of ports is controlled by what is known as an *IO block* in KSafe terminology. Except for ports that represent callback data channels or system signals delivered via callback methods (e.g., the `scan` signal), all ports should have a corresponding IO block.

In Figure 8, we have shown the BDD describing the internal structure of the SFC driver. The `SFCPTDataIOPoint` ports are controlled by an IO block of type `InhibitIOBlock` and the `GenIOIndexIOPoint` ports – by an IO block of type `MBRegIOBlock`.

Ports (points in AIM terminology) and IO blocks respectively extend the `IOPoint` and `IOBlock` abstract concepts in the AIM framework package.

Guidelines on Structural Elaboration of KSafe Drivers

The process for modelling the basic structure of a KSafe driver is as follows[4]:

1. Define a BDD initially including the driver block from the context diagram (Section 4.1) and the port data types specified during the architectural elaboration of the driver (Section 4.2).

2. For each driver port type from the IBDs of Section 4.2, that is not a callback port type:
   a. Define an IO block type.

   b. Establish composition links from the driver block to the IO block type (defined in 2.a.), and from the IO block type to the port type in question.

   c. Add the necessary multiplicity constraints to the composition links, if known. These constraints specify how many instances (minimum, maximum) of an IO block type the driver can have, and how many instances of a port an IO block type can manage. These multiplicity constraints can be added at later stages of development as well (e.g., when decisions about driver's performance are being made).

   - **Note**: IO block types and (non-callback) port types must respectively be specializations of `IOBlock` and `IOPoint` defined by the AIM Framework.

4.4. Specify Driver's Activities and their Allocations

The goal of this step is to specify the activities to be performed by the driver and state how these activities are distributed over the driver's blocks. In Figure 9, we provide a hierarchical decomposition tree for the activities of the SFC driver expressed as a BDD.

The immediate descendents of the root node (`SFC Overall`) represent the driver's main activities. These activities are the creation and deletion of a driver instance (`Create SFC`

[4] As stated earlier, further elaboration of the driver's structure will be done in a subsequent step (see Section 4.5).

`IO Driver`, `Delete SFC IO Driver`), handling of callback requests from the AIM IO Manager for reading or updating the driver parameters (`Process IO Manager Request`), alarm monitoring (`Monitor Alarm`), and the driver's core functionality (`Relay Signal Transfer`), which is transferring inhibit data from the input modules to a thorn fire central panel. This last activity is decomposed into three finer-grained activities:

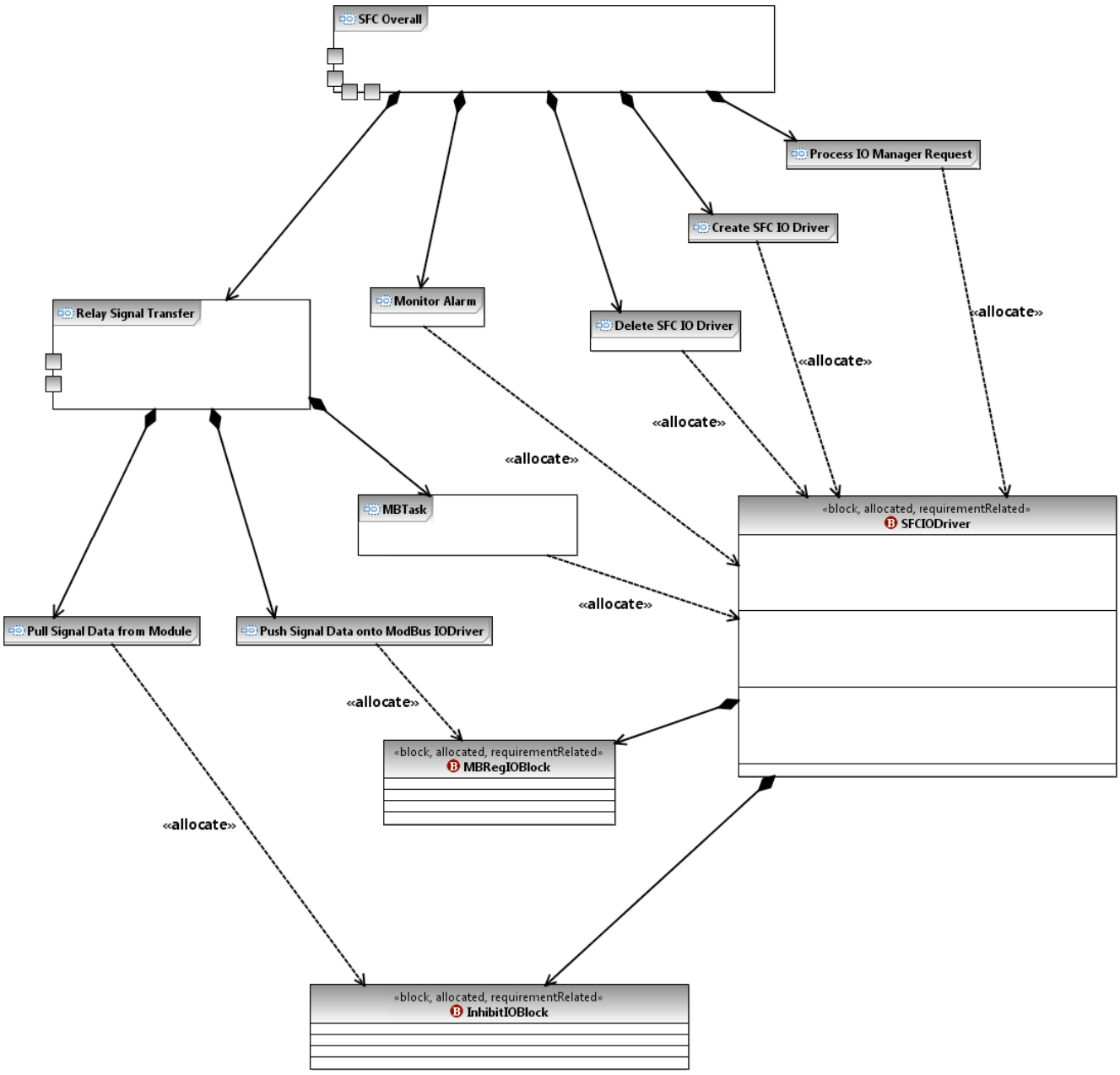

FIGURE 9: BREAKDOWN OF SFC DRIVER'S ACTIVITIES AND ALLOCATION OF ACTIVITIES TO BLOCKS

(1) Reading the inhibit data from the input modules (`Pull Signal Data from Module`), (2) forwarding the inhibit data to the ModBus driver (`Push Signal Data onto ModBus`

`IO Driver`), and (3) adjusting the rate at which data is sent to the ModBus driver (`MBTask`).

Being able to adjust the send rate is necessary because the Thorn Fire Central panel is not synchronized by the AIM framework's `scan` signal. Sending data to the device at the `scan` rate may result in an overflow and may unnecessarily overload the RCU. Therefore, the output rate from the SFC driver needs to be slowed down. This is a common design constraint in situations where a KSafe application needs to interface with an asynchronous device.

The leaf activities in the decomposition tree are *allocated* to the structural blocks of the driver. The allocation of an activity to a block means that the activity is to be fulfilled by the operations of that block. To allow for the allocations to be made, leaf-level activities should be fine-grained enough to be fulfilled by an individual block of the BDD developed earlier in Section 4.3. More precisely, a leaf-level activity should be small enough to be implementable by one or more operations in a one block. We do not decompose activities past block operations. In other words, the finest-grained activity possible is one whole block operation.

As the result of activity decomposition, some of the block operations can be directly decided. For example, the creation and deletion activities in Figure 9 give rise to two operations in the `SFCIODriver` block. But not all the operations might be known until the detailed behavioural elaboration in Section 4.5 is complete. For example, the `Process IO Manager Request` activity will be realized through several operations in the `SFCIODriver` block. These operations can be fully defined only after the configurable parameters of the driver are determined during the behavioural design in Section 4.5.

Guidelines on Identifying Driver Activities and Allocating Them

The process for developing an activity decomposition tree and allocating the activities to the driver blocks is as follows:

1. Specify the top-level activities of the driver. These should at least include: (1) lifecycle activities for creating and deletion of driver instances; (2) high-level activity(ies) to enable the reading and modification of driver parameters via AIM framework callbacks; (3) an alarm monitoring activity to satisfy the communication interface requirements of the AIM framework[5]; and (4) one or more activities capturing the core functions of the driver (for SFC, the activity representing the core function was `Relay Signal Transfer`).

2. Iteratively decompose the top-level activities in Step 1. The specifics of the decomposition will vary depending on how the driver is structured and is thus

[5] Depending on the driver's purpose, the alarm monitoring activity may or may not be implemented. For example, SFC does not implement alarm monitoring.

based mainly on human judgment. Generally, decomposition should be necessary only for the core functions. The activity breakdown we provide for `Relay Signal Transfer` in Figure 9 captures the essence of what one might expect in other drivers: there are sub-activities for *reading* from ports, *writing* to ports, and ensuring proper *synchronization* and read/write *rates*.

- **Note:** The interactions between activities are not modelled in the activity decomposition tree, nor are concurrency and sequencing of activities. These will be modelled using activity diagrams, as we discuss in Section 4.5.

4.5. Elaborate the Driver's Behaviour

The goal of this step is to describe the detailed behaviour of the driver. During this step, block operations are decided as well and added to the structural model constructed earlier in Section 4.3.

SysML offers two different notations for describing the behavioural aspects of a system. These are state diagrams and activity diagrams. A state diagram shows the possible set of states a block instance can be in and how the instance transitions between different states in response to internal events or events received from the environment. State diagrams are typically not developed for all system blocks, but rather only for those with internal states (or modes of operation) and complex transitions between these states.

Activity diagrams are variants of state diagrams. They concentrate on the flow of control and data between block instances not in response to events but rather as a result of internal processing. Activity diagrams bear a lot of similarity to flowcharts and are particularly useful for expressing work performed in the implementation of an operation.

Both state diagrams and activity diagrams may be needed for expressing the behaviours of KSafe drivers. In this report, we only illustrate the use of activity diagrams because the data relay behavior of SFC is stateless and therefore state diagrams are not applicable here. For drivers with stateful behaviours, modeling the states and the transitions between them is an important part of the development process. We are currently investigating the design of other KSafe drivers so that we can provide representative examples of state diagrams within the KSafe domain and augment our methodology with guidelines about state modeling. For a general introduction to state modeling in SysML, see [3].

Figure 10 through Figure 13 respectively show the activity diagrams for `SFC Overall`, `Relay Signal Transfer`, `Create SFC IO Driver`, and `Process IO Manger Request`. These activities were part of the activity decomposition tree in Figure 9.[6]

Each activity in an activity diagram can have input and output parameters. The inputs and outputs of an activity are shown using parameter nodes (similar to port nodes attached to blocks). An action may not start before the required inputs arrive and it may not terminate before its outputs are produced. However, if an owning activity is terminated, its contained activities are terminated too.

In Figure 10, we can see the sequencing and interactions between the immediate descendant activities of `SFC Overall`. The process begins with the creation of an SFC driver instance. Once an instance has been created, three parallel activities begin: processing the IO Manager requests, relaying the signals read from the `SFCPTDataIOPoint` ports to the `GenIOIndexIOPoint` ports, and monitoring for alarms. These activities do not terminate until a `delete` signal is delivered by the IO Manager, upon which the delete activity is executed.

The activity diagram in Figure 11 describes the behavior of `Relay Signal Transfer`. Initially, the non-terminating activity named `MBTask` starts to execute. After a configurable delay time, `MBTask` reads the data stored in an internal mailbox, called `SFCMailbox`[7]. It then processes the data, and writes the processed data to a second mailbox called `MBRegMailbox`.

The content stored in `SFCMailbox` is produced by the `Pull Signal Data from Module` activity and consumed by the `MBTask` activity. `MBTask` in turn serves as the producer of content for `MBRegMailbox`. The content of this latter mailbox is consumed by the `Push Signal Data onto ModBus IO Driver` activity. Both `Pull Signal Data from Module` and `Push Signal Data onto ModBus IO Driver` are triggered by the periodical `scan` signal delivered by the IO Manager.

The use of intermediate mailboxes provides a flexible way for queuing the messages and synchronizing the tasks. The structure of the data entries stored in `SFCMailbox` and `MBRegMailbox` is specified through two new blocks, named `SFCMBDataToIOBlock` and `SFCDataInhibit`. These new blocks are added to the structural model of the driver along with the required dependencies to the blocks (see Figure 14).

[6] Activity diagrams for all the activities shown in Figure 9 and the descendants of these activities at the level of individual blocks are provided in the appendix.

[7] Mailboxes are special types of resources for data storage. In our models, we denote this by the «`datastore`» stereotype.

Finally, Figure 12 and Figure 13 show how a driver instance is created and how the callback requests are processed. The former activity is described using a set of actions (atomic activities), and the latter is further broken down into smaller activities.

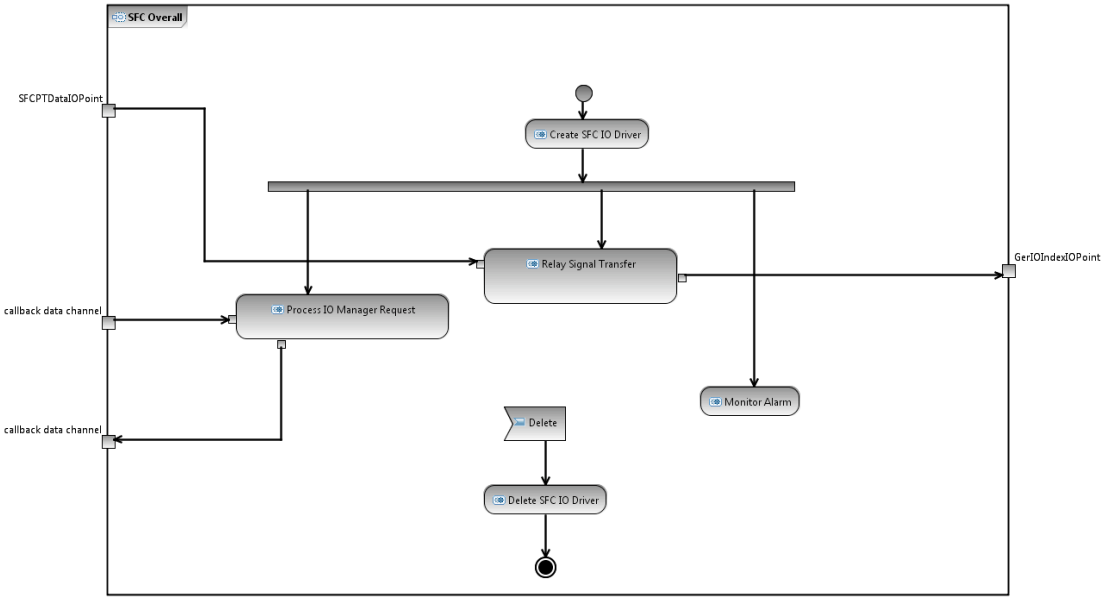

FIGURE 10: BEHAVIOURAL DESIGN OF "SFC OVERALL" ACTIVITY

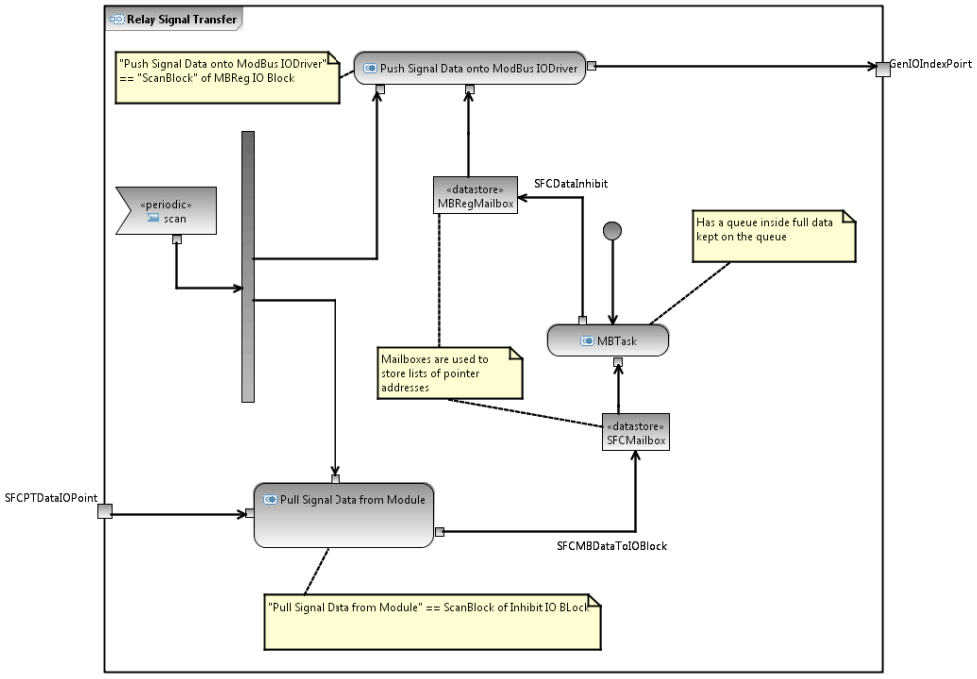

FIGURE 11: BEHAVIOURAL DESIGN OF "RELAY SIGNAL TRANSFER" ACTIVITY

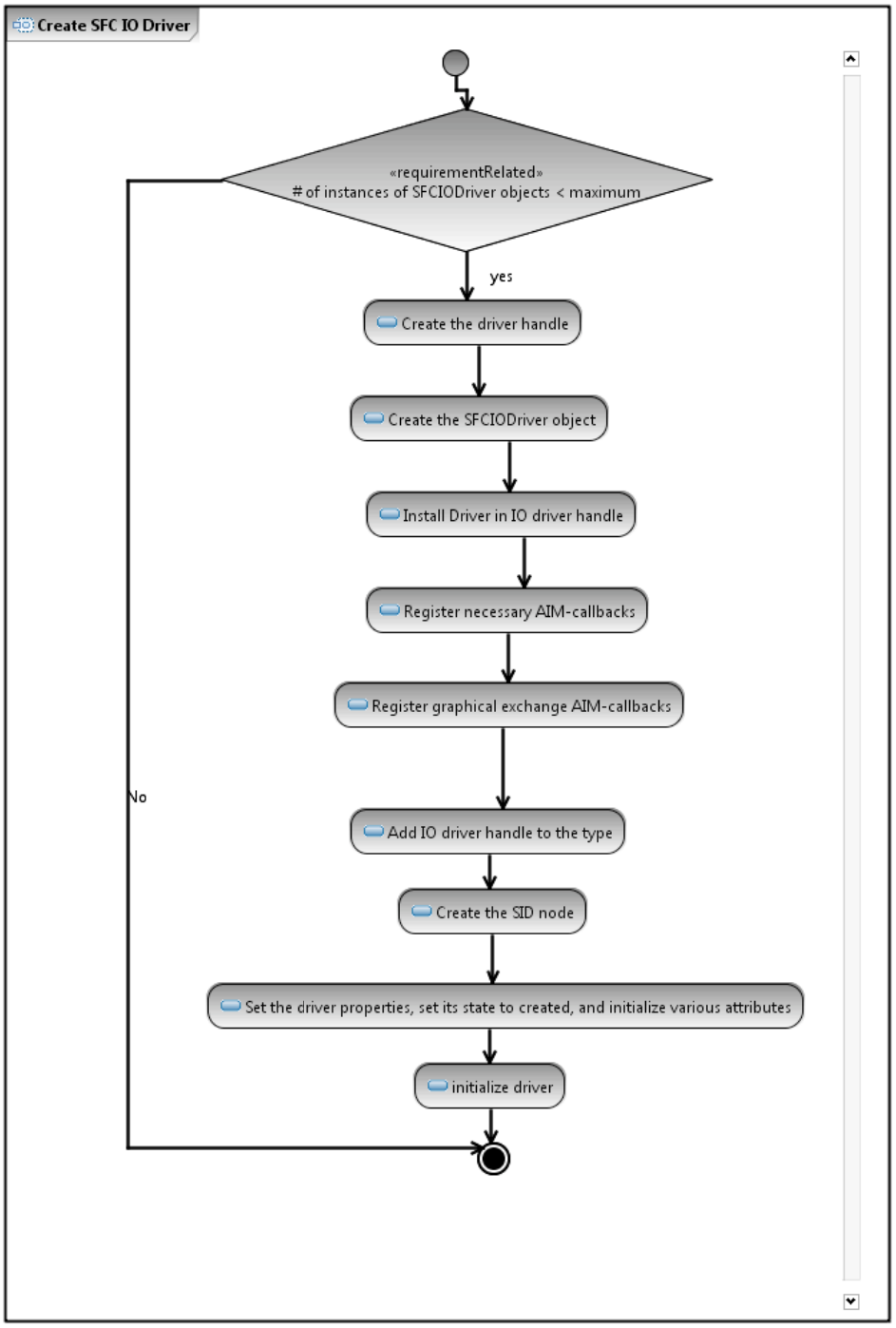

FIGURE 12: BEHAVIOURAL DESIGN OF "CREATE SFC IO DRIVER" ACTIVITY

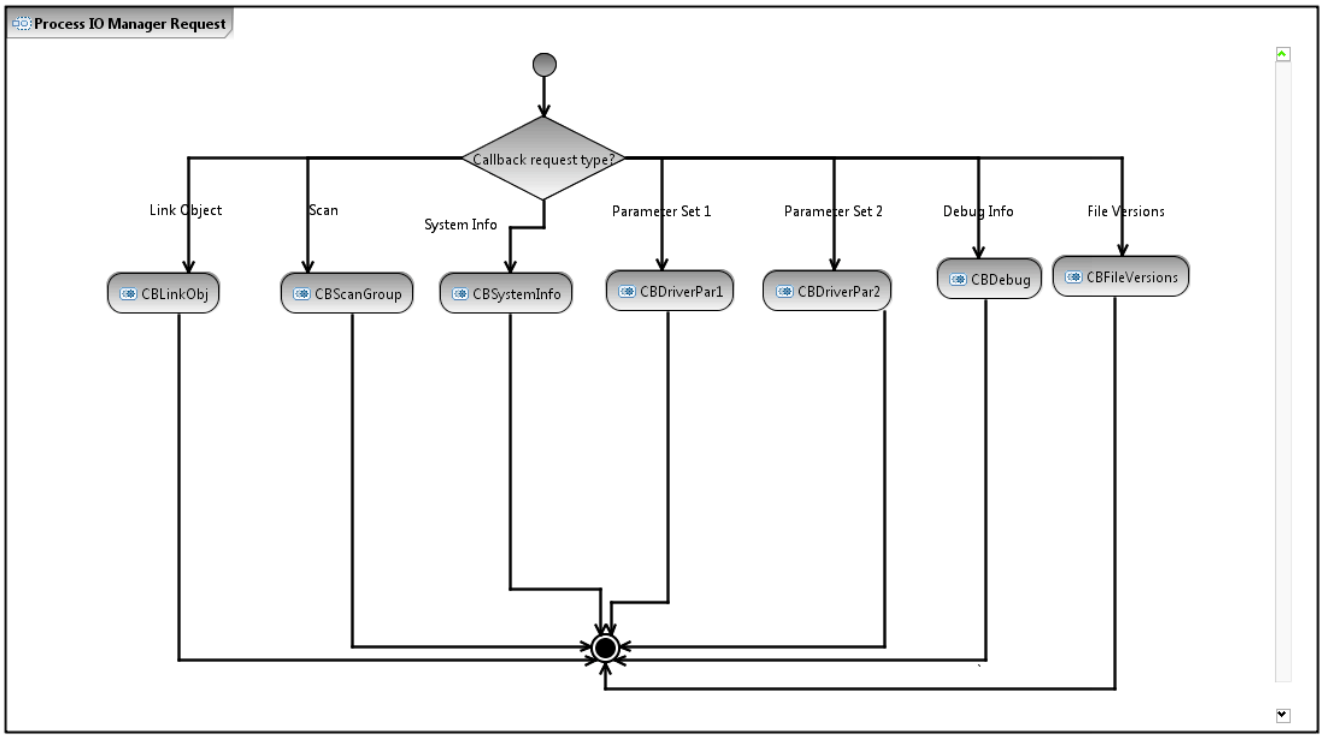

FIGURE 13: BEHAVIOURAL DESIGN OF "PROCESS IO MANAGER REQUEST" ACTIVITY

As the result of behavioural elaboration, we will arrive at a set of activities that are small enough to be implemented as block operations. In particular, if an activity can be defined in terms of only actions, it will be captured as a block operation (such as the creation activity in Figure 12). Higher-level activities can be turned into block operations as well using these more basic operations, but in the SFC example, this was not needed. Also, note that actions are more abstract than actual program statements and are not meant to provide the same information as code. From our experience inspecting the SFC driver code, we have noticed that the comment items that describe what each major code block in a program method does are at a good level of abstraction for being modeled as actions.

Figure 14 shows the structural model for the SFC driver refined with new knowledge from the behavioural design phase. Particularly, the diagram contains new blocks (namely, `SFCMBDataToIOBlock` and `SFCDataInhibit`) that enable communication between the different activities, and block operations for realizing the intended behaviour of the driver.

Guidelines on Behavioural Elaboration of KSafe Drivers

The overall process for behavioural elaboration is as follows:

1. Define the behavior of each composite activity in terms of its sub-activities. Continue until an activity can be defined only using actions, at which point the activity is defined as a basic block operation.

2. Update the structural design with the block operations identified and any new blocks needed for inter-activity communications (in the SFC driver, these blocks were `SFCMBDataToIOBlock` and `SFCDataInhibit`).

To maintain quality and consistency in behavioural design, the following rules need to be considered:

- The activity diagram for a composite activity should involve all the sub-activities of that activity (defined in the activity decomposition tree). This ensures that all lower-level activities and actions remain reachable from the root activity in the activity decomposition tree.

- The input and output of each activity must be specified using parameter nodes. For the root activity in the decomposition tree (Section 4.4), the parameter nodes correspond to the ports defined in the architectural view(s) (Section 4.2).
    - **Note:** System signals are more suitably modeled as events *within* the activity diagrams rather than activity parameter nodes. For example, see how the `scan` signal is modeled in Figure 11.

What we described in Sections 4.1 through 4.5 covers the construction of design models for a KSafe driver. In Section 4.6, we are going to explain how these design models can be precisely linked to the underlying requirements for the driver.

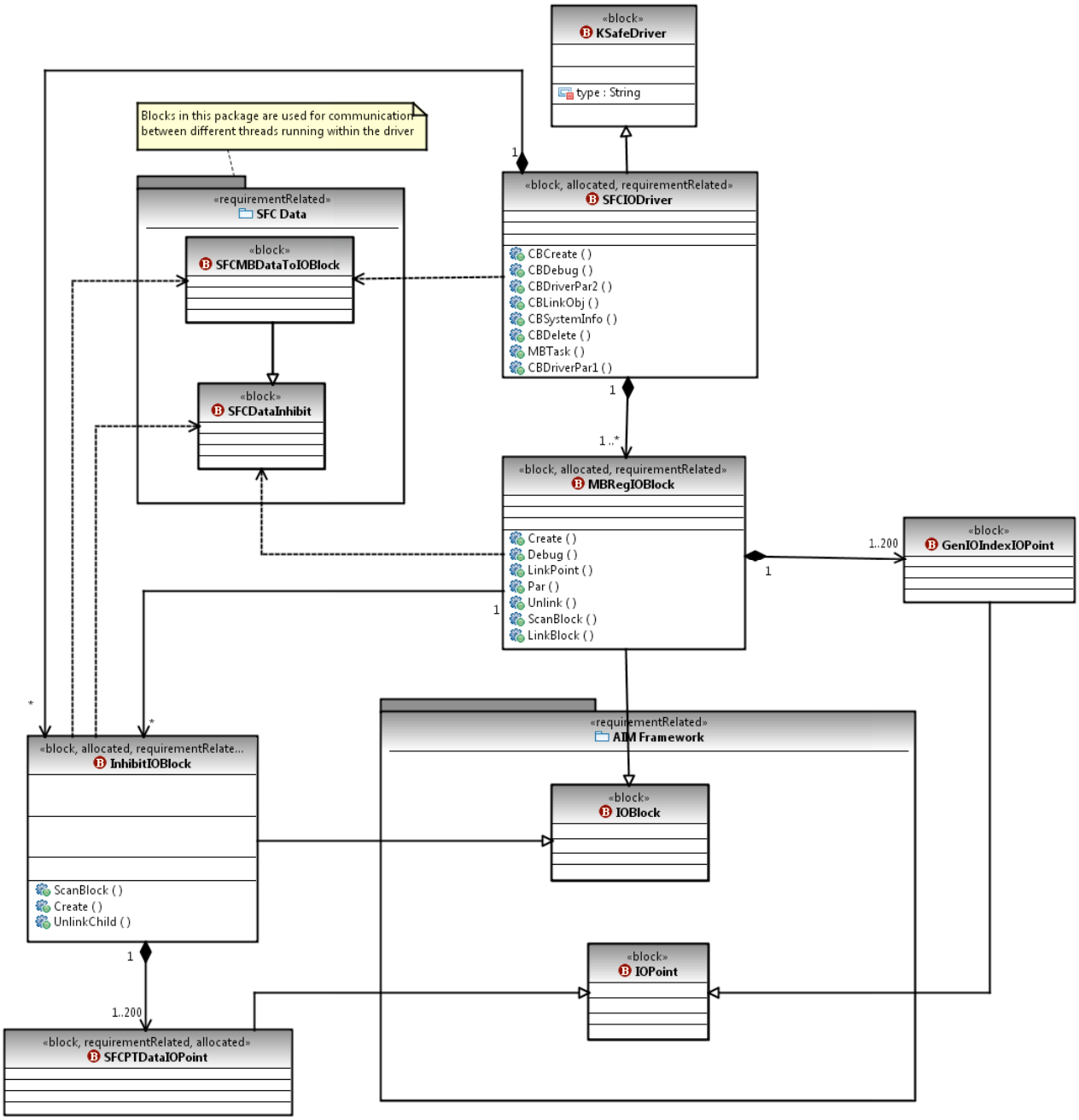

FIGURE 14: STRUCTURAL MODEL COMPLETED BASED ON INFORMATION FROM BEHAVIOURAL DESIGN

4.6. Link the Requirements and Design

In this step, we establish traceability links between the driver's requirements and its design using SysML Requirements Diagrams. The traceability links specify which parts of the design contribute to the satisfaction of each requirement. We use three general patterns, shown in Figure 15, for defining the links.

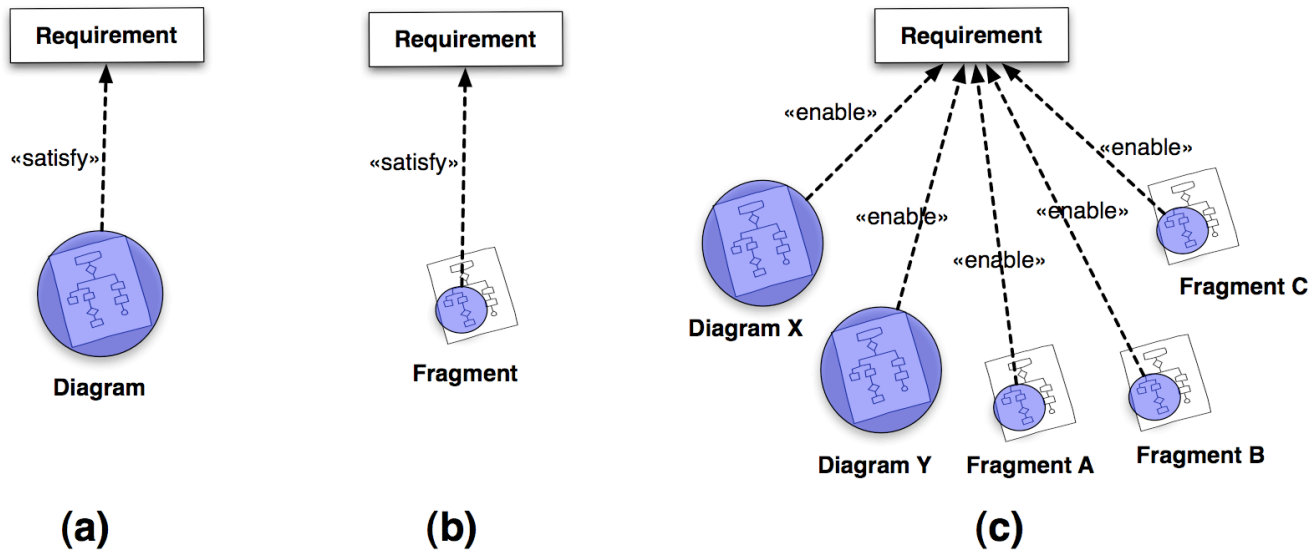

FIGURE 15: PATTERNS FOR TRACEABILITY LINKS BETWEEN THE REQUIREMENTS AND THE DESIGN

In the first case, (a), a complete diagram (usually an activity diagram) is connected to a requirement using a «`satisfy`» link. This means that the design elements in the diagram fully satisfy the requirement and there is no need for linking further evidence from the design to argue that the requirement is properly addressed. An example is shown in Figure 16, where the `Pull Signal Data From Module` activity is specified as satisfying the requirement for manual triggering of data transmission. The allocation of this activity to the `InhibitIOBlock` was made during the activity decomposition step in Section 4.4 and the allocation link is shown here only for clarity.

As one can see from Figure 16, we also maintain a direct link, labelled «`trace`», from the requirement to the block(s) that contribute to its satisfaction. The «`trace`» links can be inferred from the «`satisfy`» and «`allocate`» links; but explicating them simplifies impact analysis tasks and is thus beneficial.

The second case, (b) in Figure 15, is the same as case (a) except that a diagram fragment, as opposed to an entire diagram, provides relevant evidence for the satisfaction of a requirement. Figure 17 shows an example of such a situation. The requirement in the diagram of Figure 17 states that it should be possible to turn the debug messages on or off. This requirement is satisfied by the fragment of the `CBDebug` activity that enables the debug flag to be changed. The satisfaction link is hence made from the relevant fragment of the activity.

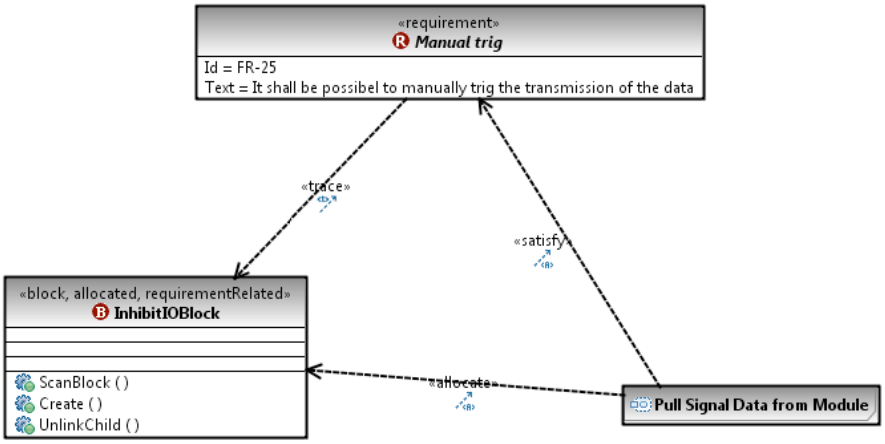

FIGURE 16: A SATISFACTION LINK FROM A COMPLETE ACTIVITY (DIAGRAM) TO A REQUIREMENT

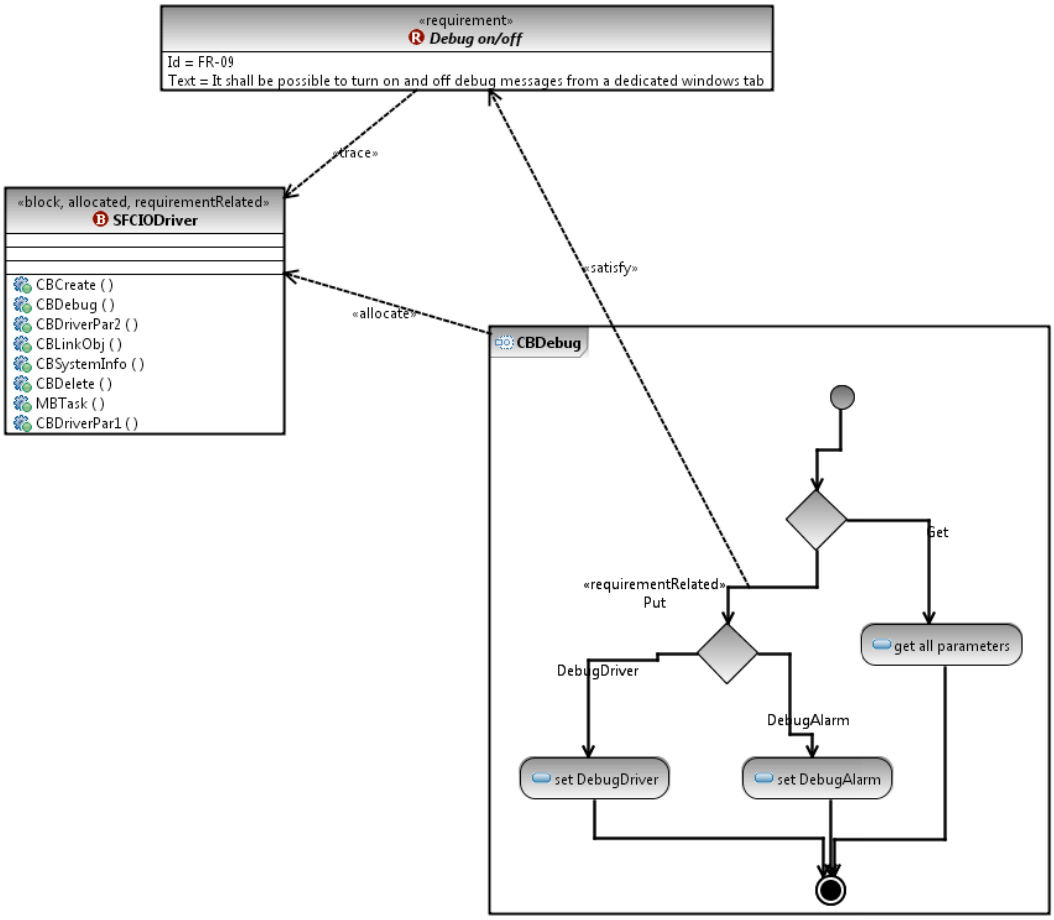

FIGURE 17: SATISFACTION LINK FROM A FRAGMENT OF AN ACTIVITY DIAGRAM TO A REQUIREMENT

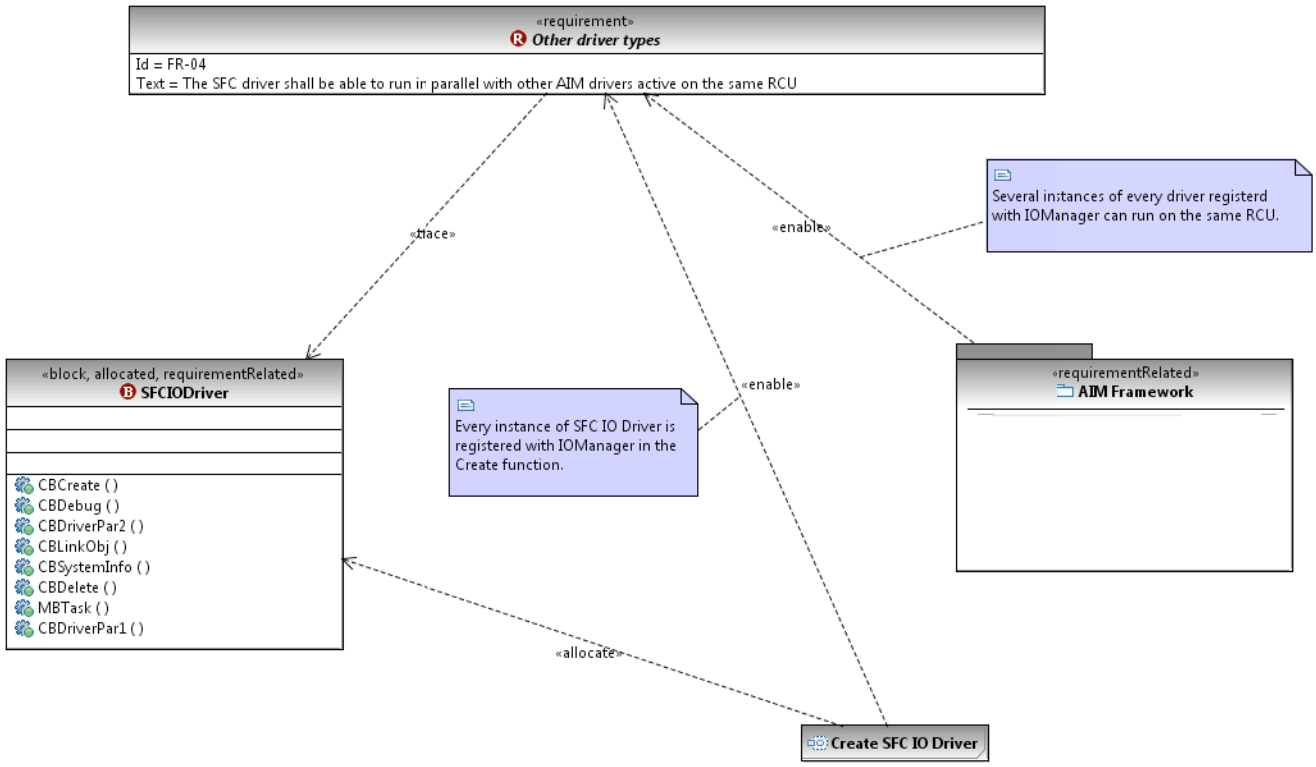

FIGURE 18: ENABLING LINKS FROM MULTIPLE DESIGN DIAGRAMS TO A REQUIREMENT

In the final case, (c) in Figure 15, we need to deal with a situation where no single design diagram contains all the evidence necessary to show the satisfaction of a requirement and the evidence is distributed over multiple diagrams. In such a case, we link all the relevant pieces of evidence (diagrams and/or diagram fragments) to the requirement in question using «`enable`» links.

In Figure 18 we show an example for case (c). The requirement in the figure states that it shall be possible to run SFC in parallel with other AIM drivers running on the same RCU. The AIM framework has built-in mechanisms for parallelizing different drivers on the same RCU and for specifying the multiplicities and the types of drivers, but for these mechanisms to work, the drivers have to register with AIM. Hence, the satisfaction of the requirement in the diagram of Figure 18 will depend both on proper registration by the SFC driver during creation as well as the runtime facilities provided by AIM. The details of the AIM framework are outside the scope of our analysis; therefore, the «`enable`» link in Figure 18 is made from the AIM Framework package (which is a fragment of the driver's structural diagram).

5. Observations

The work described in this report was prompted in part by the observations we made during the certification meetings between KM and TÜV. In Table 2, we list some of the recurring issues raised at these meetings and the certifiers' proposed resolutions.

TABLE 2: SOME RECURRING ISSUES RAISED DURING CERTIFICATION AND PROPOSED RESOLUTIONS

No.	Issue	Proposed Resolution (by Certifier)
1	Relationships between the software blocks and whether they are at the same or different levels (subsystem, component, module) are not clear.	Develop a diagram to show the decomposition of the software system into its constituent blocks.
2	Interactions between components and subsystems have not been stated precisely.	A more detailed description of the architecture must be developed. The interfaces between function modules, input and output bits for terminals, and logical information flows between terminals must be specified.
3	The semantics of diagrams in the documentation are ambiguous.	Elements on each diagram need to be clearly labeled and the meaning of each box and arrow needs to be specified.
4	Application workflows are difficult to understand from the (textual) descriptions provided.	A sequence diagram or an activity diagram would be very helpful for easier understanding of the workflows.
5	High-level design is presented using nested lists of (textual) bullet items. Certifier had difficulty grasping an overall view of the system from these lists.	A diagram (at a high level of abstraction) must be included instead of the bullet lists.
6	Different modes of the system and admissible transitions between different modes not documented.	Different states in the system, and the conditions and events for moving from one mode to another must be captured using a state machine.

In our assessment, a large fraction of the difficulties in safety certification arise due to the use of text-based documents. Text is highly prone to ambiguity, incompleteness, and redundancy. This leads to the suppliers and certifiers having to invest a substantial amount of time and human resources resolving the ambiguities, identifying and

addressing the areas of incompleteness, and ensuring that overlapping information across multiple documents remains consistent. Using diagrammatic illustrations that are not represented in standardized notations (or their extensions) helps little to address the problem, because the semantics of the diagrams would be unknown and never stated explicitly. Hence, the diagrams could well become another source of ambiguity, incompleteness, and redundancy (in particular see issue 3 in Table 2).

Further, text is very difficult to query and manipulate automatically. In particular, although the majority of the information necessary for safety certification naturally results from regular development activities (requirement, design, and verification and validation), it takes developers significant manual effort to extract the safety-relevant information from the documents built during these activities, and put the information in an appropriate form.

Our position is that models, and not text-based documents, should serve as the main sources of development information – documents, when needed, should be generated from models. For the purpose of safety certification, models are beneficial in many important respects. Most notably:

1. Models can be employed to clarify the expectations of safety standards and recommended practices, and develop concrete guidelines for ISDS suppliers;

2. Models expressed in standard notations avoid the ambiguity and redundancy problems associated with text-based documentation;

3. Models provide an ideal vehicle for preserving traceability and the chain of evidence between hazards, requirements, design elements, implementation, and test cases;

4. Models present opportunities for partial or full automation of many laborious safety analysis tasks (for example, impact analysis, completeness and consistency checking, and test case generation).

Our work in this report provides a concrete example of how MDE can be applied within KM and how one can systematize the modeling process by developing a methodology according to the context of application. Using MDE is very much aligned with the expectations of the certifiers in the maritime domain too. In fact, based on our observations and as suggested by Table 2, the certifiers frequently recommend the inclusion of models as part of the safety evidence for maritime ISDSs. This is a growing trend and we anticipate that the future recommended practices by the certification bodies will place a lot more emphasis on the construction of high-quality system models.

6. Summary and Future Steps

In this report, we developed a set of methodological guidelines for constructing SysML diagrams for KSafe drivers. Our primary focus was providing support for the safety certification process. As a case study and a way to illustrate our methodology, we built detailed SysML diagrams for one of KSafe drivers, called the Safety Fire Central.

In the future, we plan to improve our methodology to account for the following:

- ***Writing better requirements:*** The quality of the design of a system or component is necessarily influenced by how accurately its requirements are stated and how well these requirements are organized. Providing guidelines on writing and classification of requirements will result in better understandability, a more accurate design, and more precise requirements-to-design links.

- ***Handling non-functional requirements:*** We limited the scope of the methodology in this report to functional requirements. We would like to generalize the work to cover non-functional requirements (such as performance and availability) as well.

- ***Modelling of state-based behaviours:*** As we discussed in Section 4.5, capturing the behavioural aspects of a system may involve both activity and state diagrams. In this report, we concentrated on behavioural modeling using activity diagrams. We are now investigating how state diagrams can be best introduced into the design of KSafe drivers.

7. References

1. Object Management Group (OMG), OMG Systems Modeling Language, The Official OMG SysML site. 2009, Object Management Group: http://www.omgsysml.org/.

2. Object Management Group (OMG). OMG Systems Modeling Language (OMG SysML), version 1.1. 2008; Available from: http://www.omg.org/docs/formal/08-11-02.pdf.

3. Friednenthal, S., A. Moore, and R. Steiner, A Practical Guide to SysML: The Systems Modeling Language. 1 ed. The MK/OMG Press Series. 2008: Morgan Kaufmann OMG Press.

8. Appendix

Below, we provide the complete set of SysML diagrams built for the Safety Fire Central (SFC) driver.

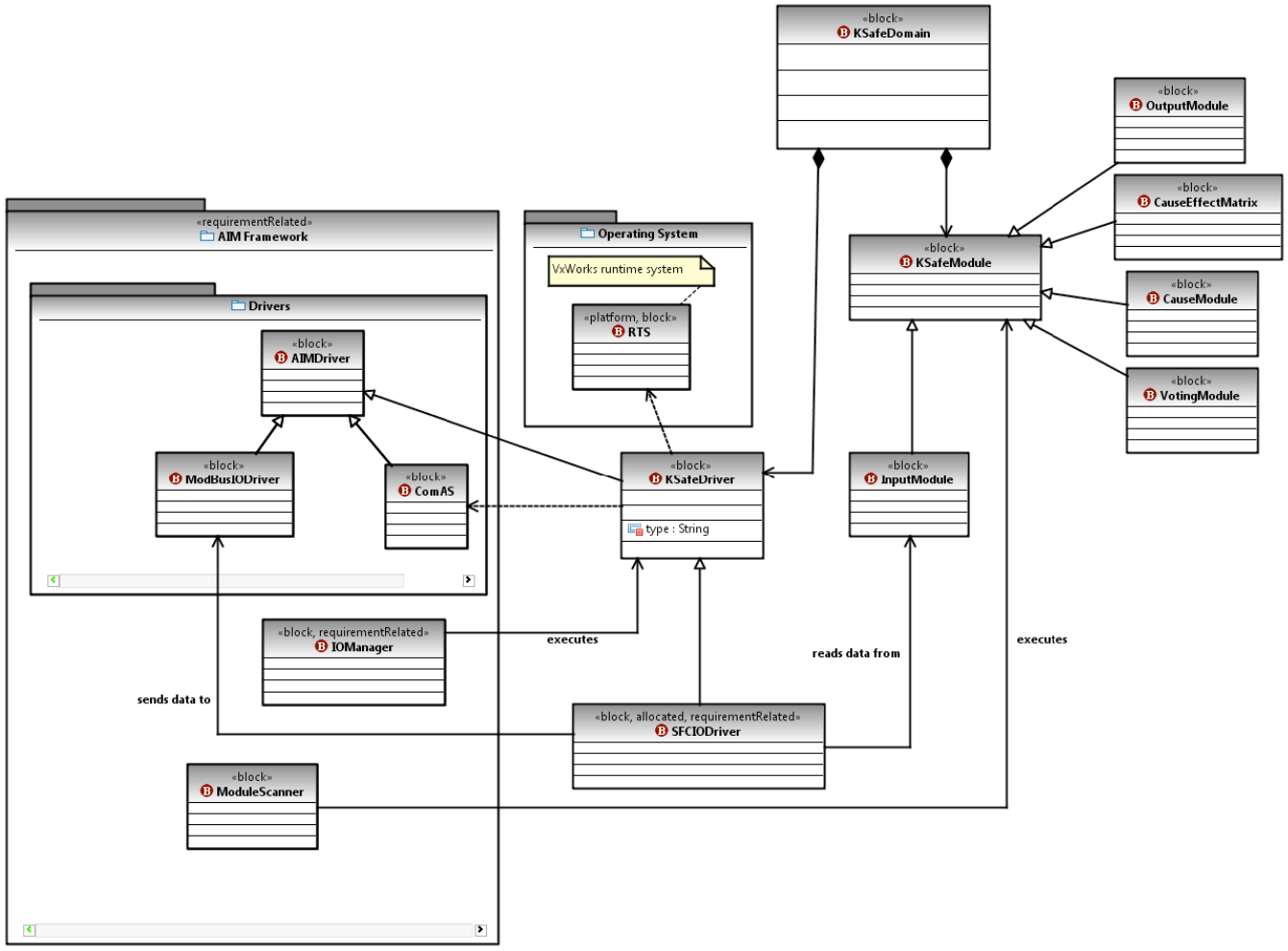

FIGURE 19: SFC DRIVER CONTEXT DIAGRAM.

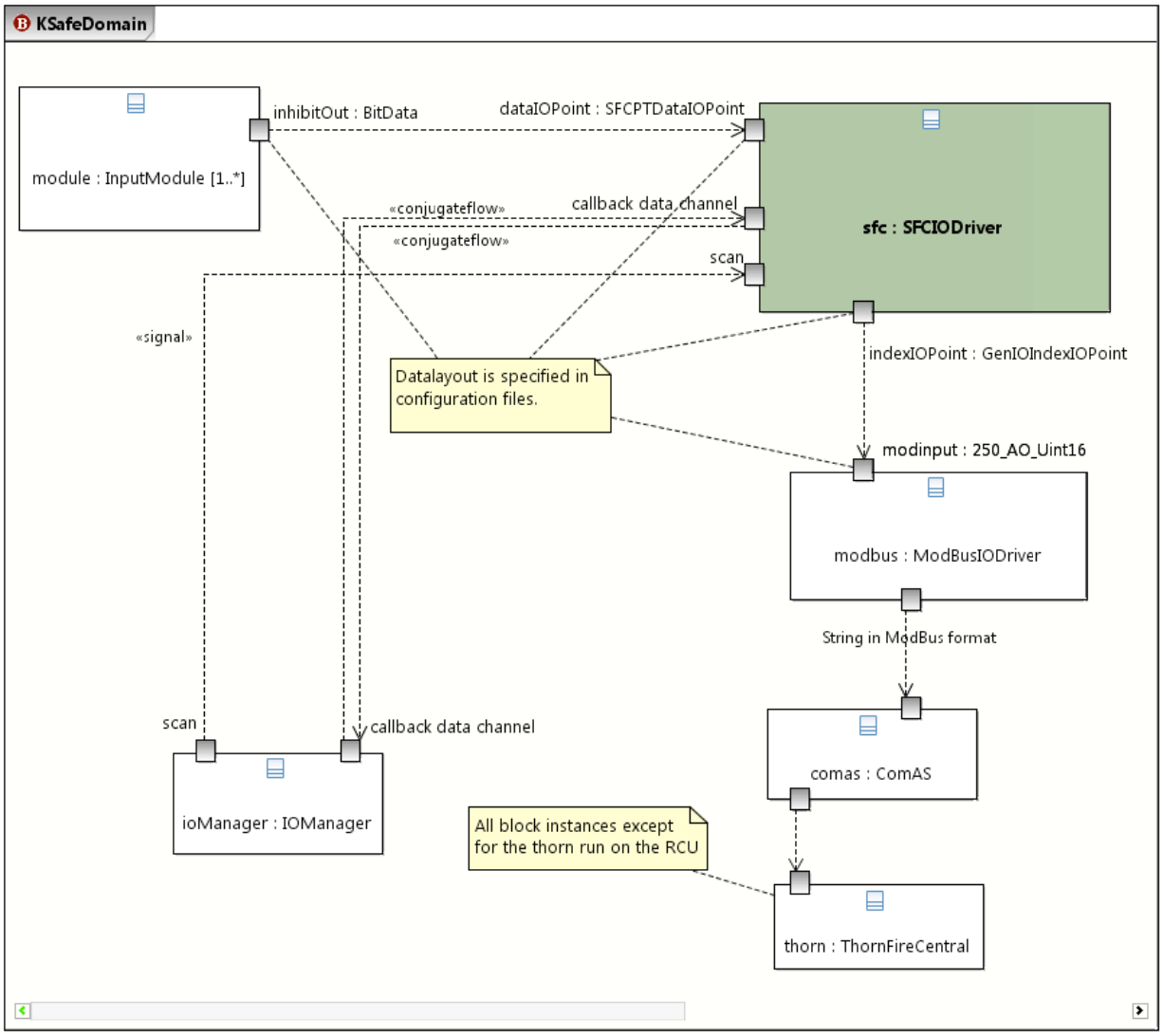

FIGURE 20: SFC DRIVER ARCHITECTURE DIAGRAM.

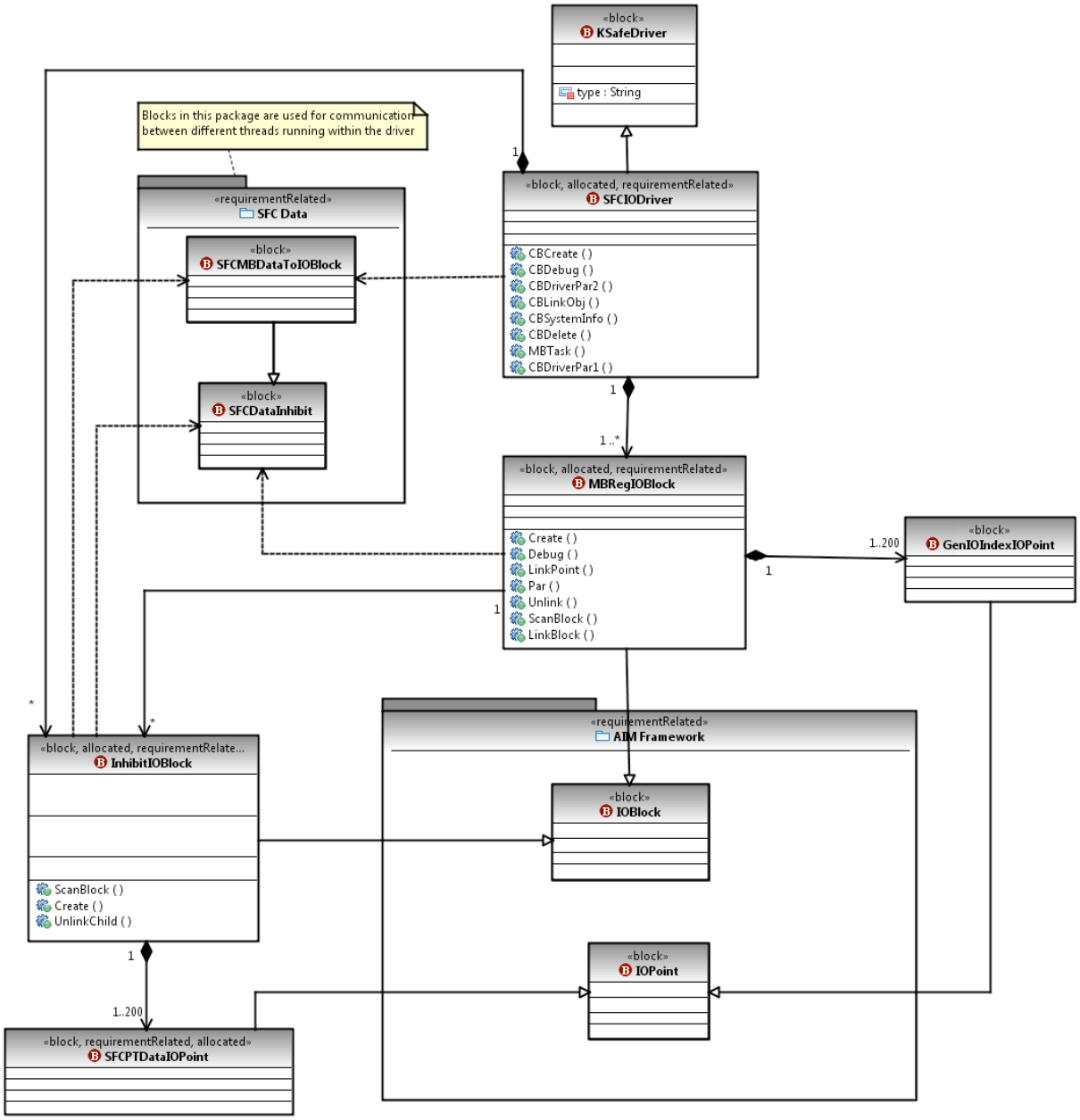

FIGURE 21: SFC DRIVER DETAILED STRUCTURAL DIAGRAM

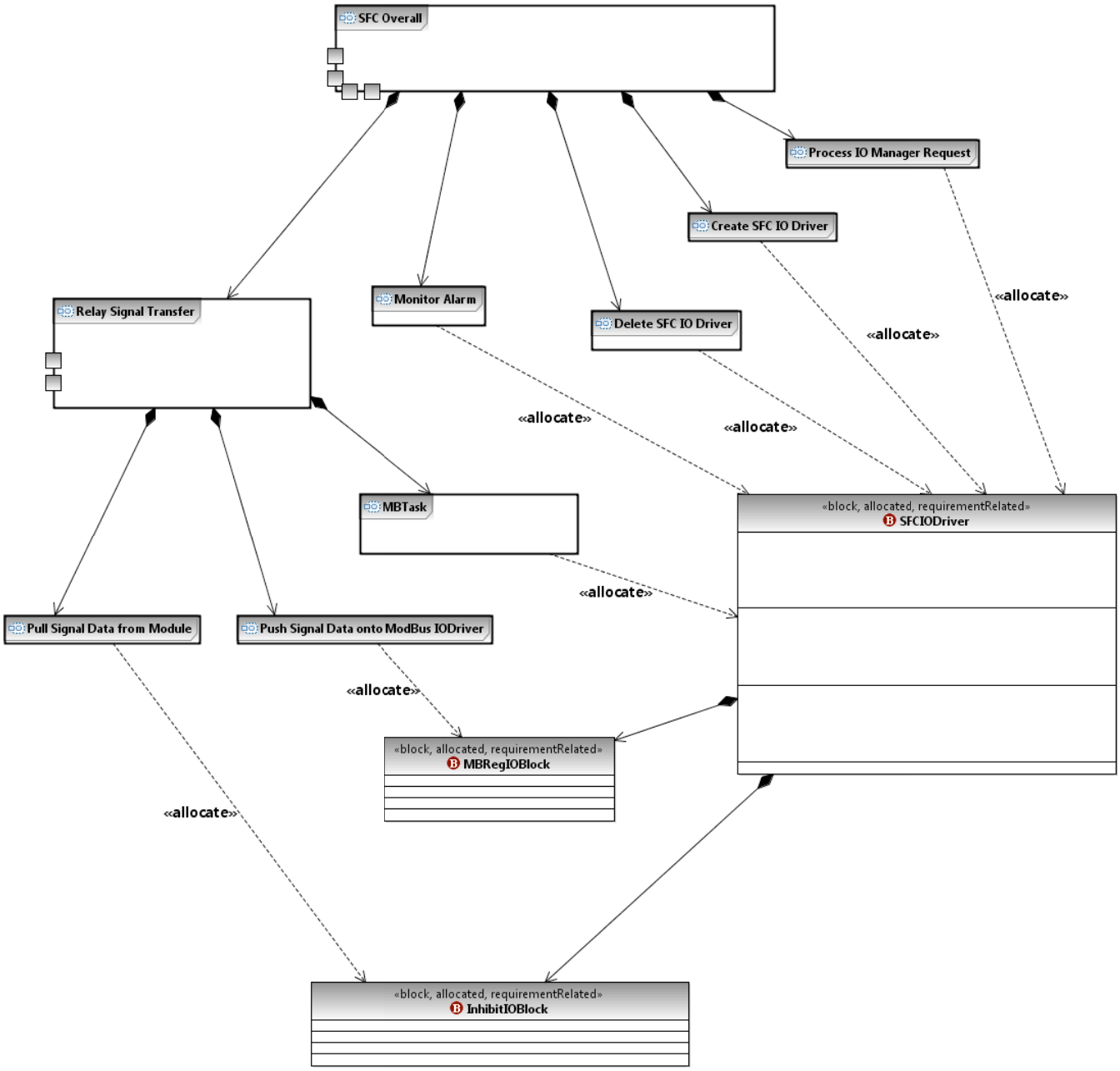

FIGURE 22: SFC DRIVER ACTIVITY DECOMPOSITION.

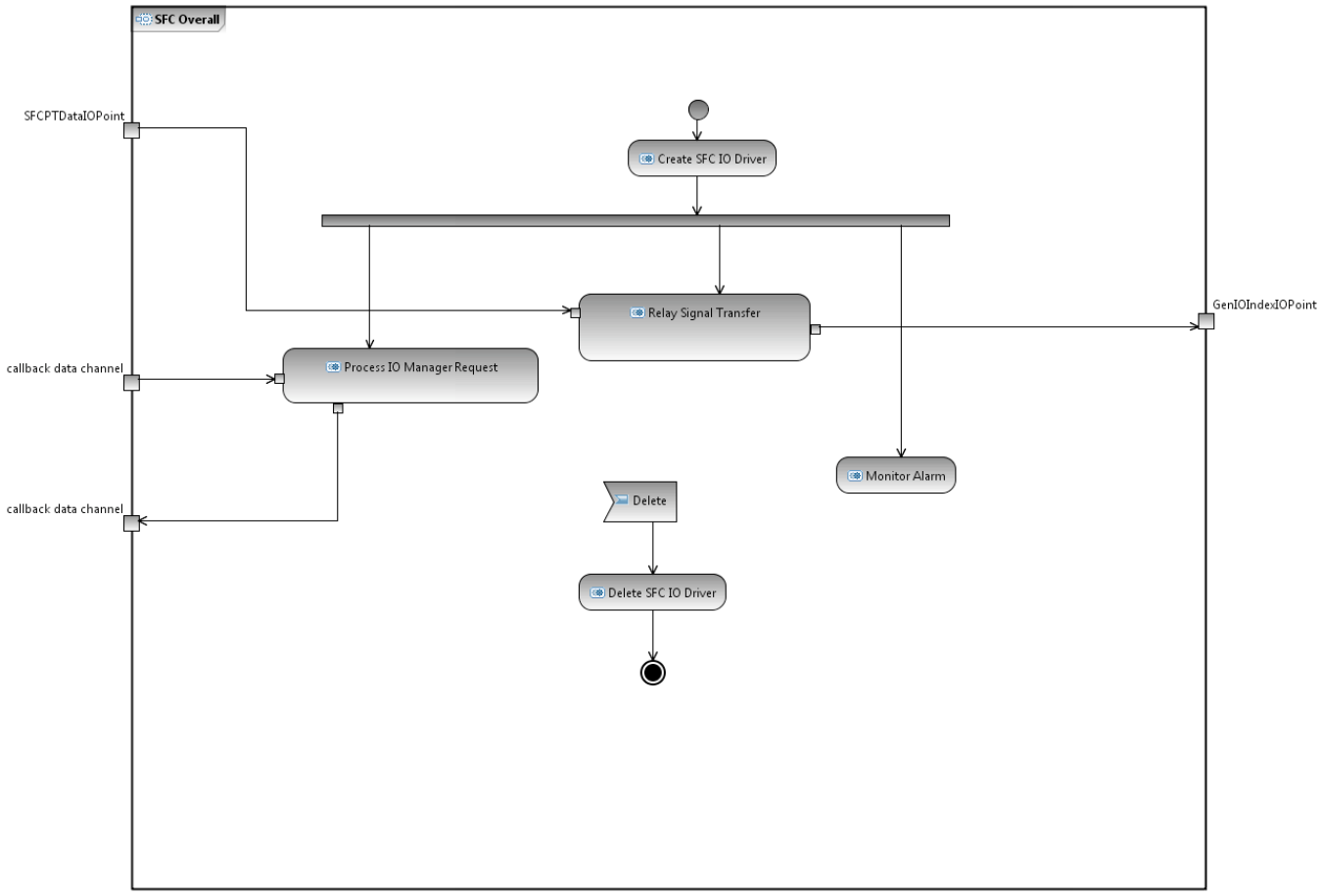

FIGURE 23: SFC DRIVER OVERALL FUNCTION ACTIVITY DIAGRAM.

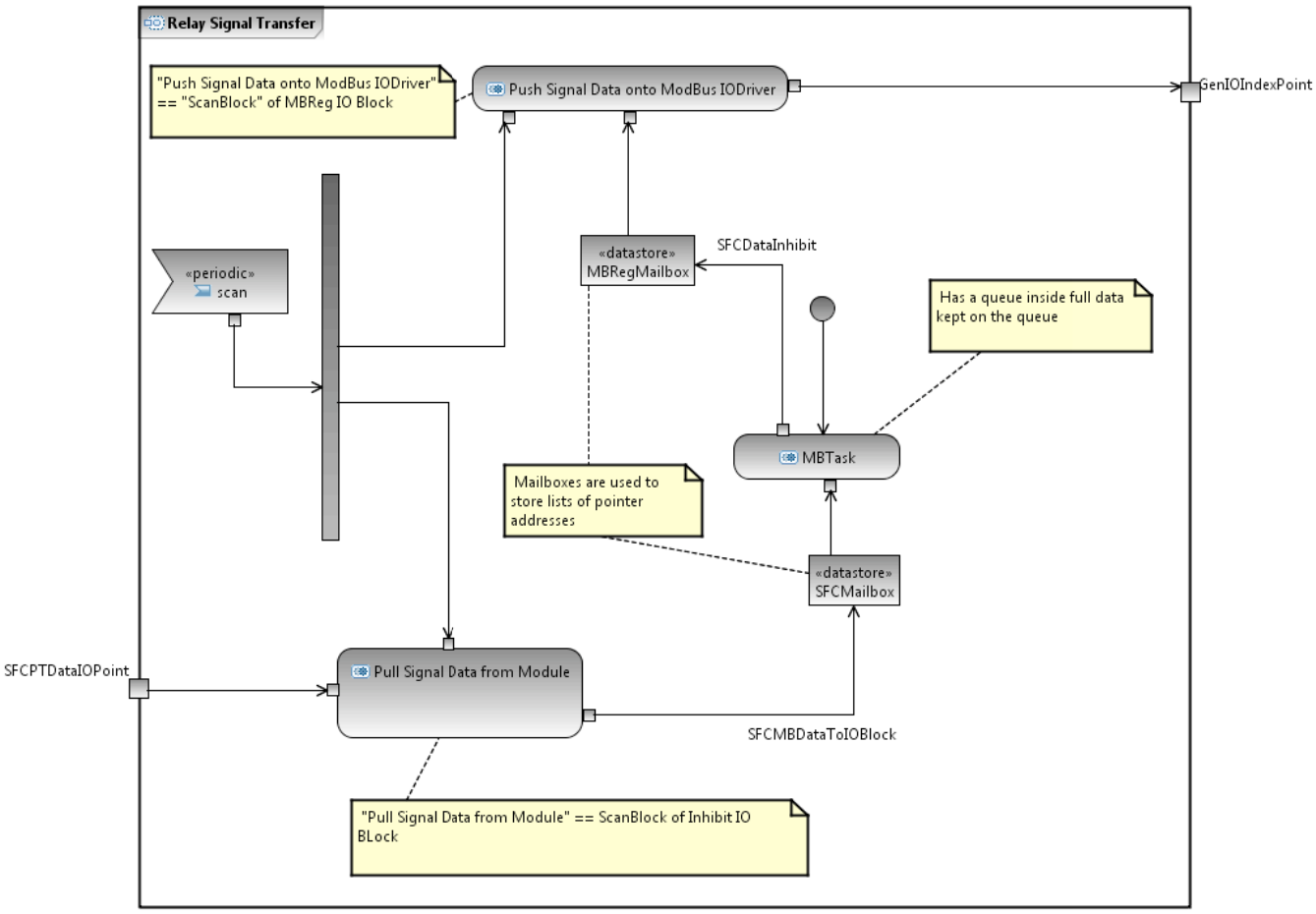

FIGURE 24:SFC DRIVER RELAY SIGNAL TRANSFER ACTIVITY DIAGRAM.

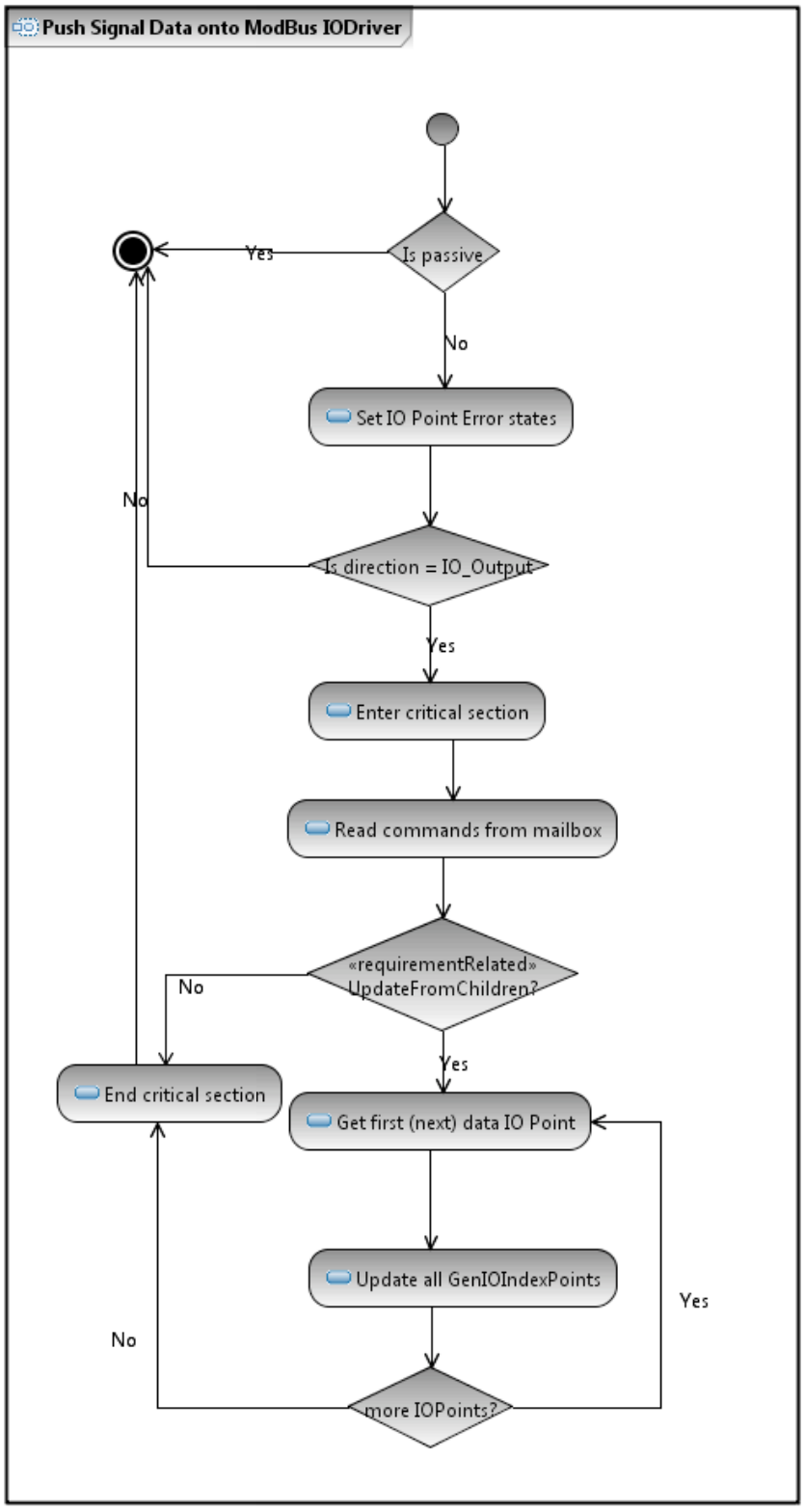

FIGURE 25: SFC DRIVER PUSH SIGNAL DATA ONTO MODBUS IODRIVER.

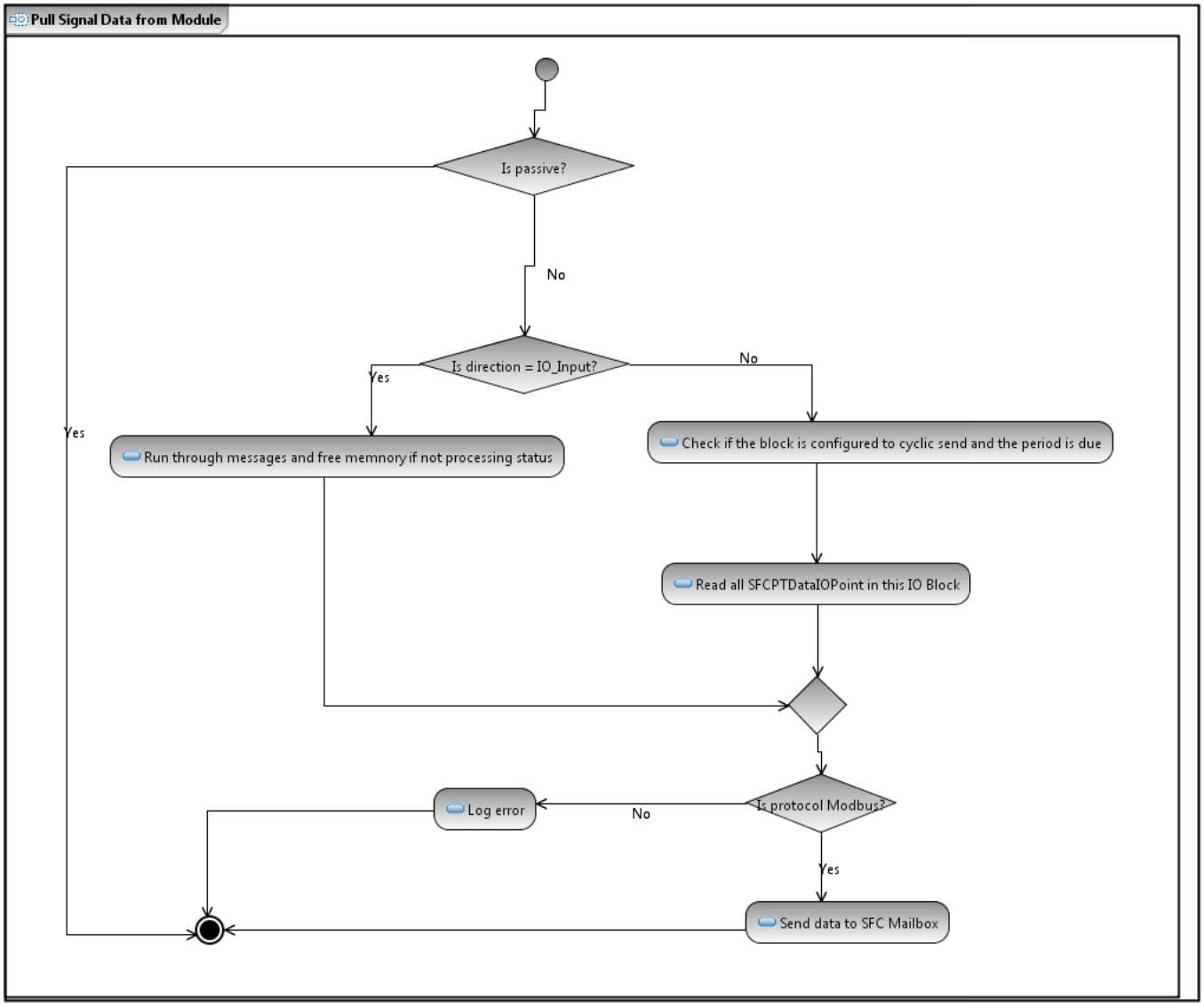

FIGURE 26: SFC DRIVER PULL SIGNAL DATA FROM MODULE ACTIVITY DIAGRAM.

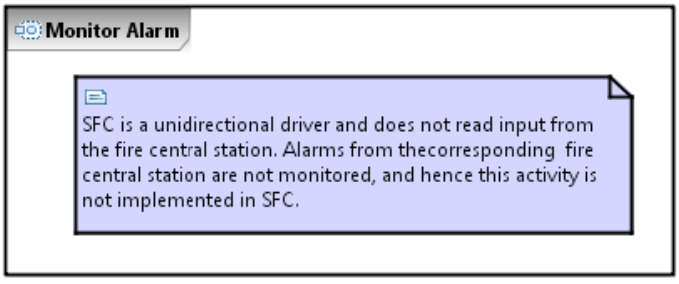

FIGURE 27: SFC DRIVER MONITOR ALARM ACTIVITY DIAGRAM.

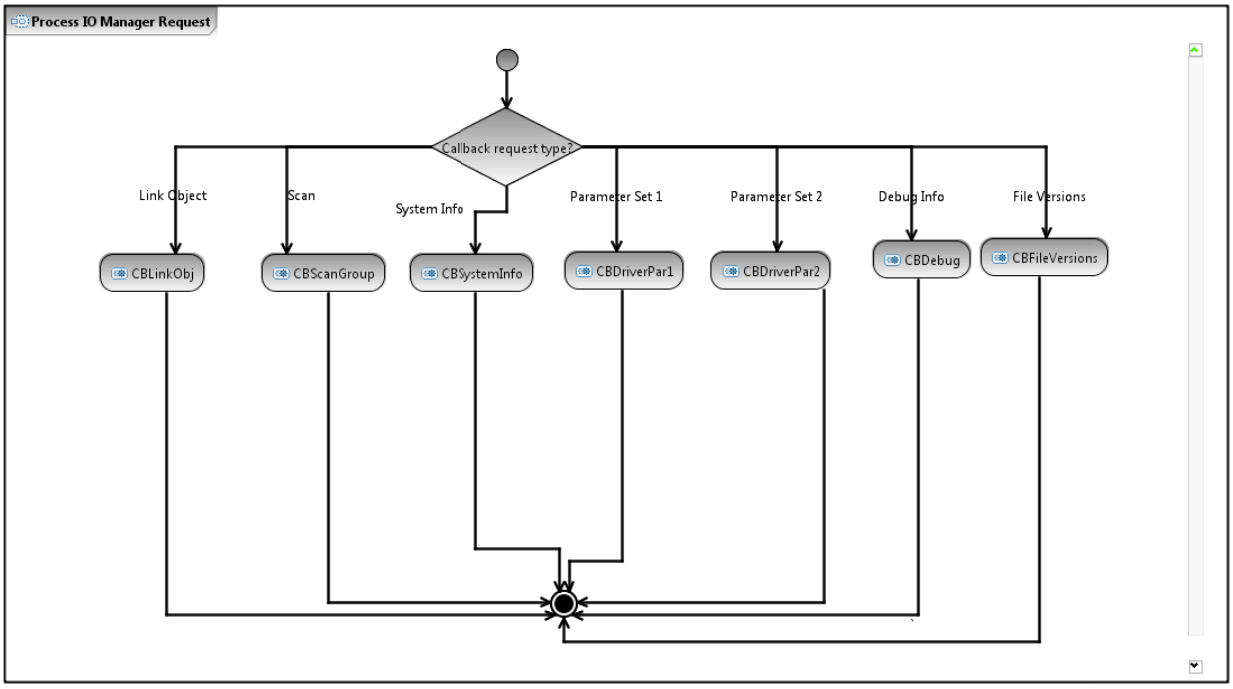

FIGURE 28: SFC DRIVER PROCESS IO MANAGER REQUEST ACTIVITY DIAGRAM.

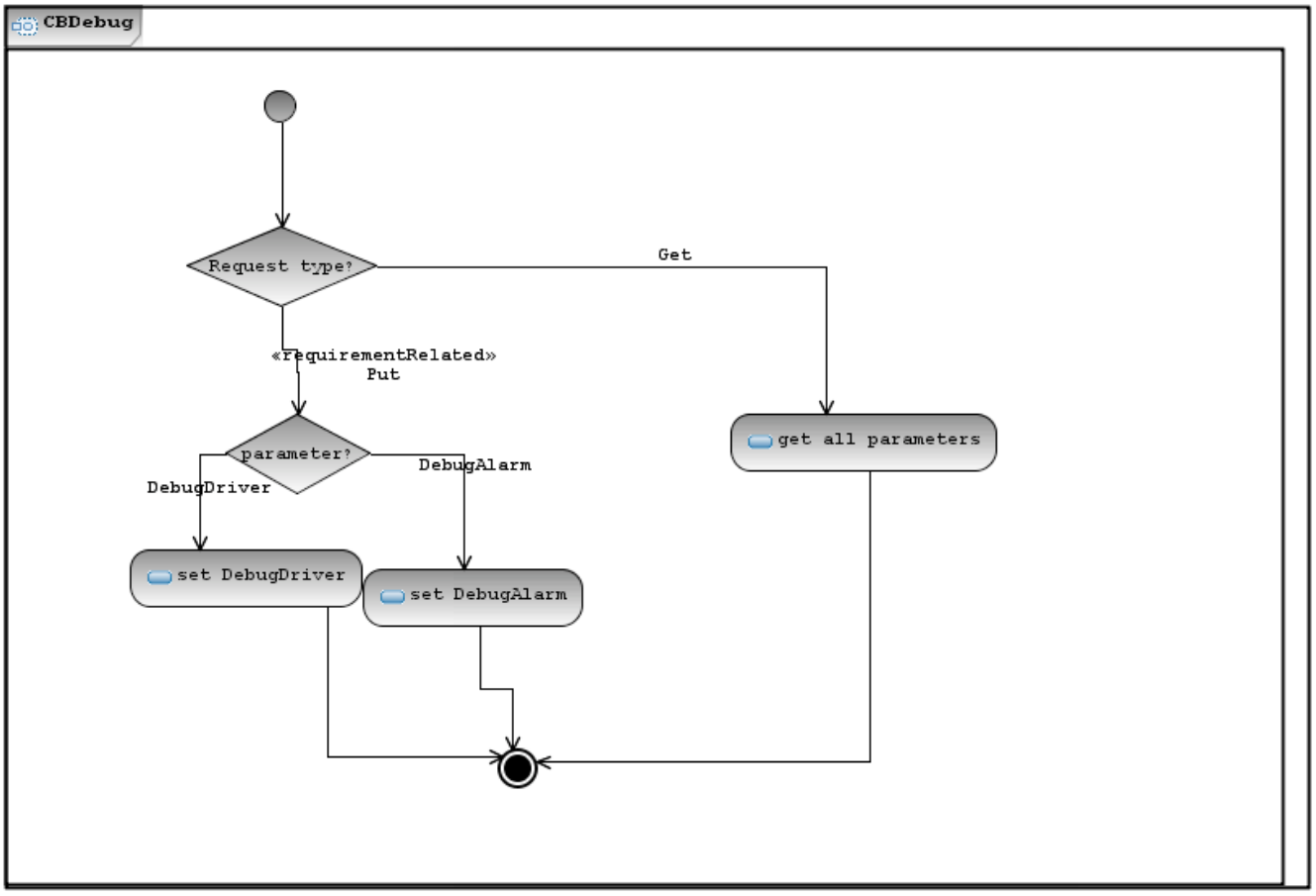

FIGURE 29: SFC DRIVER DEBUG ACTIVITY DIAGRAM.

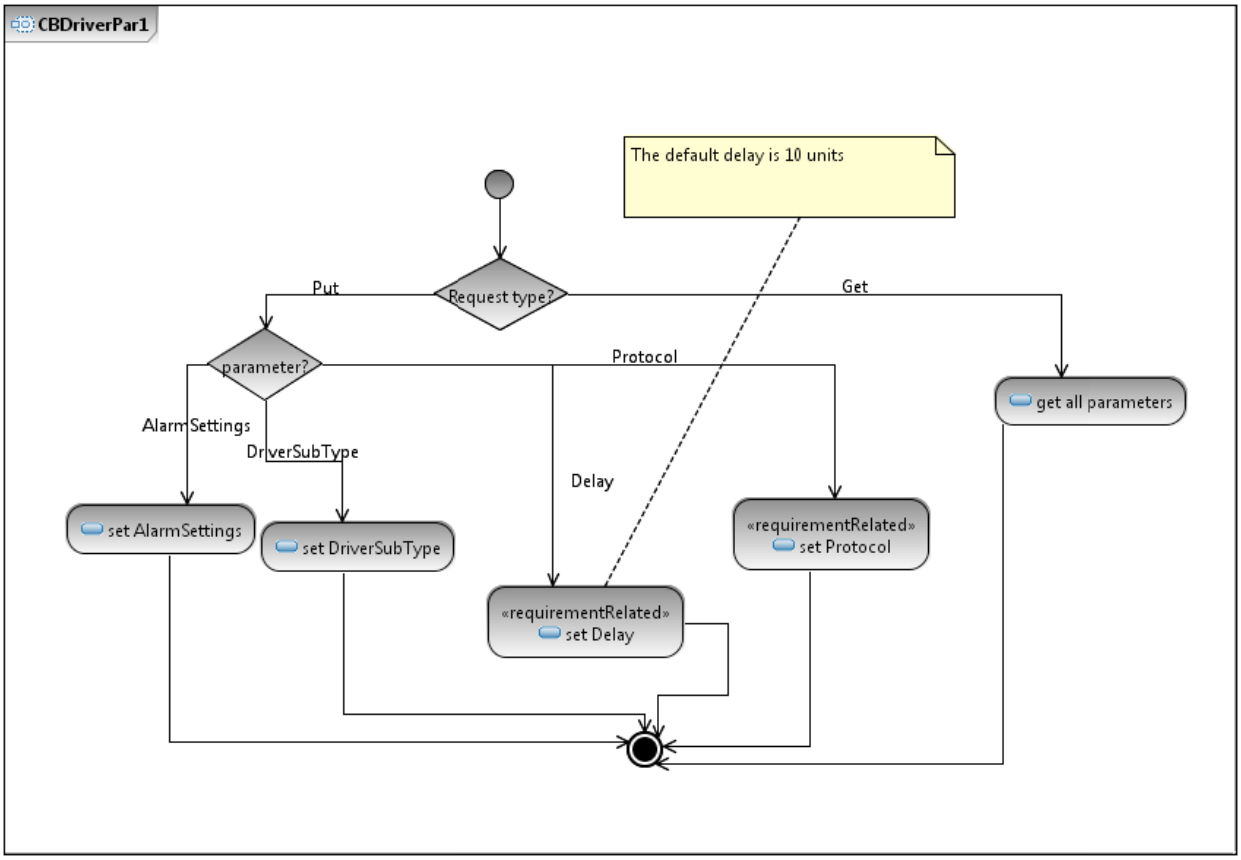

FIGURE 30: SFC DRIVER CBDRIVERPAR1 ACTIVITY DIAGRAM.

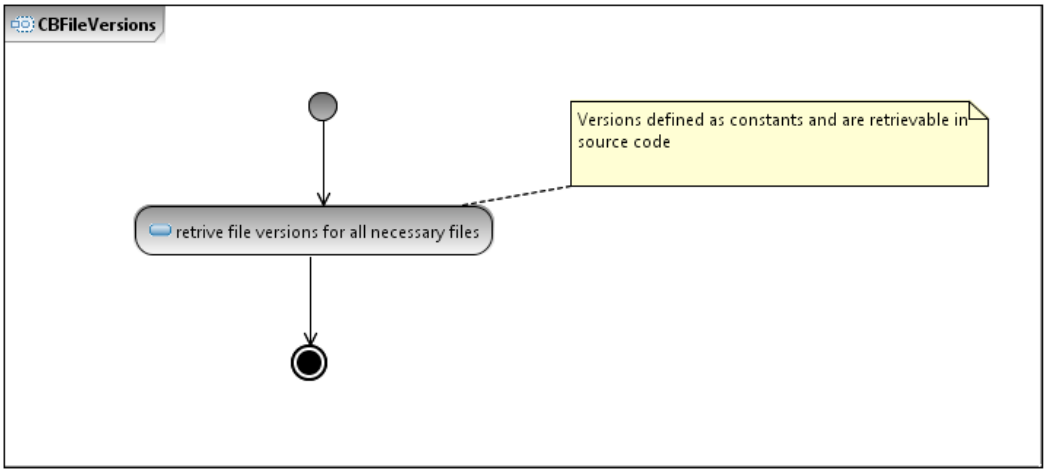

FIGURE 31: SFC DRIVER CBFILEVERSIONS ACTIVITY DIAGRAM.

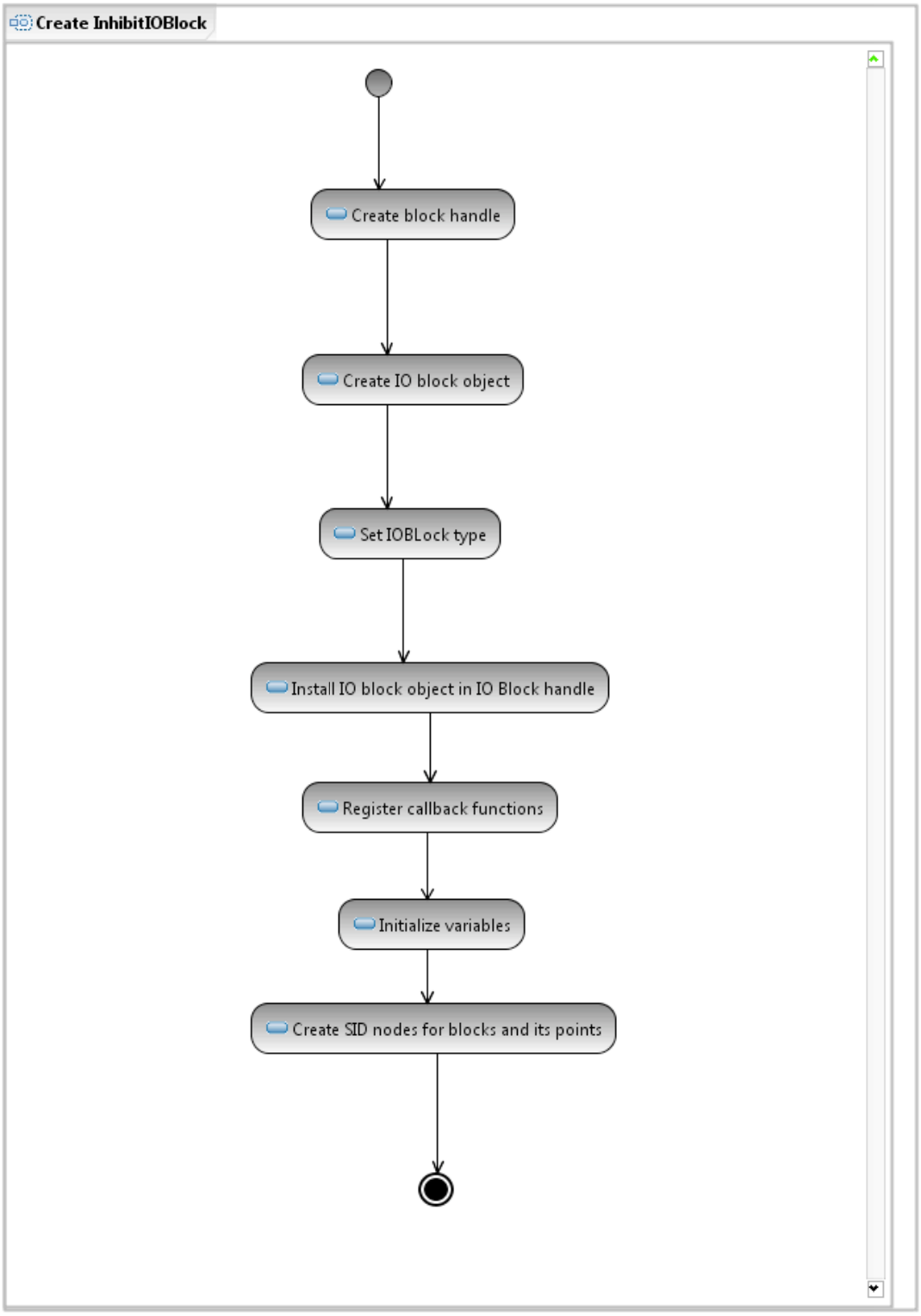

FIGURE 32: SFC DRIVER CREATE INHIBITIOBLOCK ACTIVITY DIAGRAM.

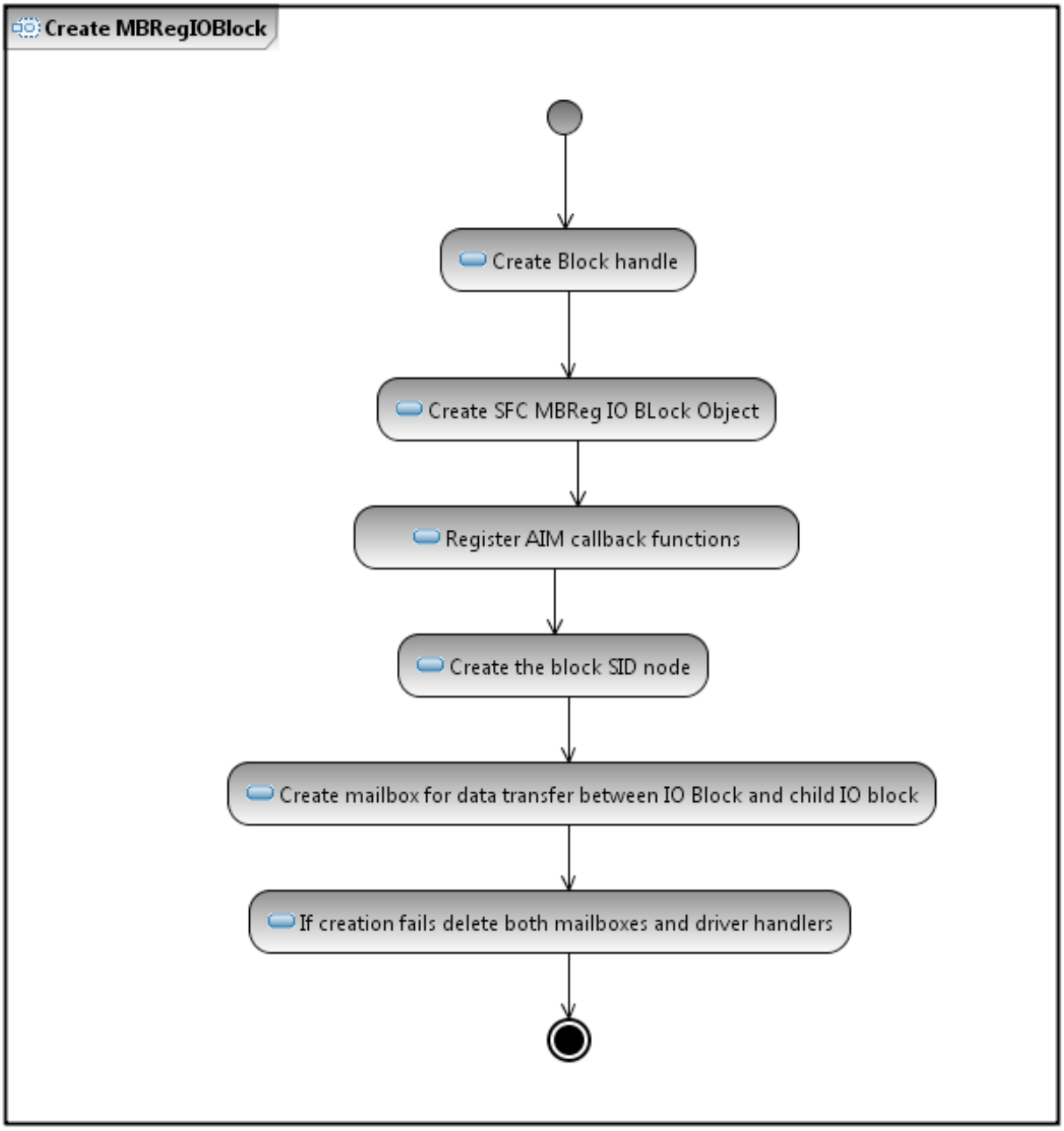

FIGURE 33: SFC DRIVER CREATE MBREGIOBLOCK ACTIVITY DIAGRAM.

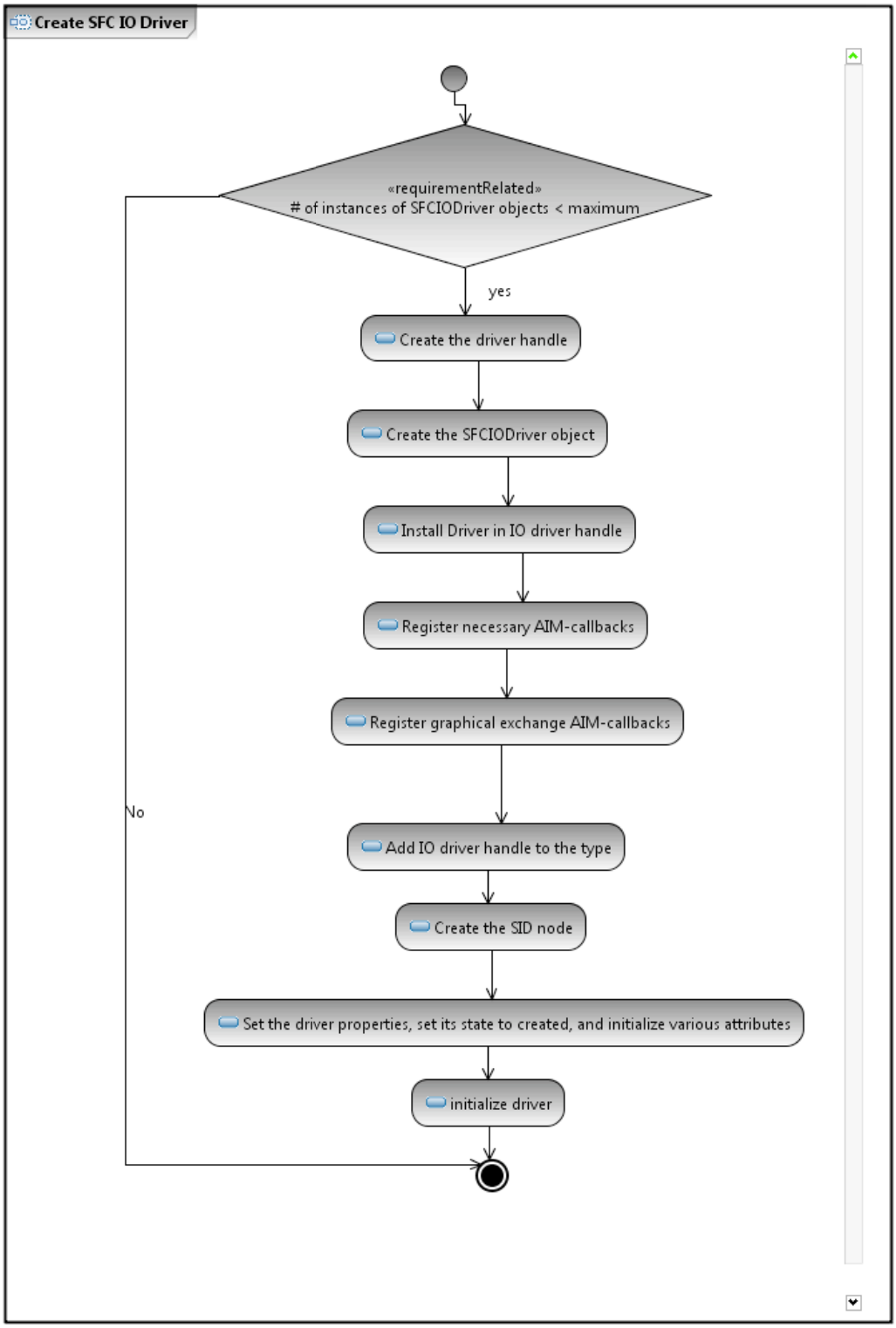

FIGURE 34: SFC DRIVER CREATE SFC IODRIVER ACTIVITY DIAGRAM.

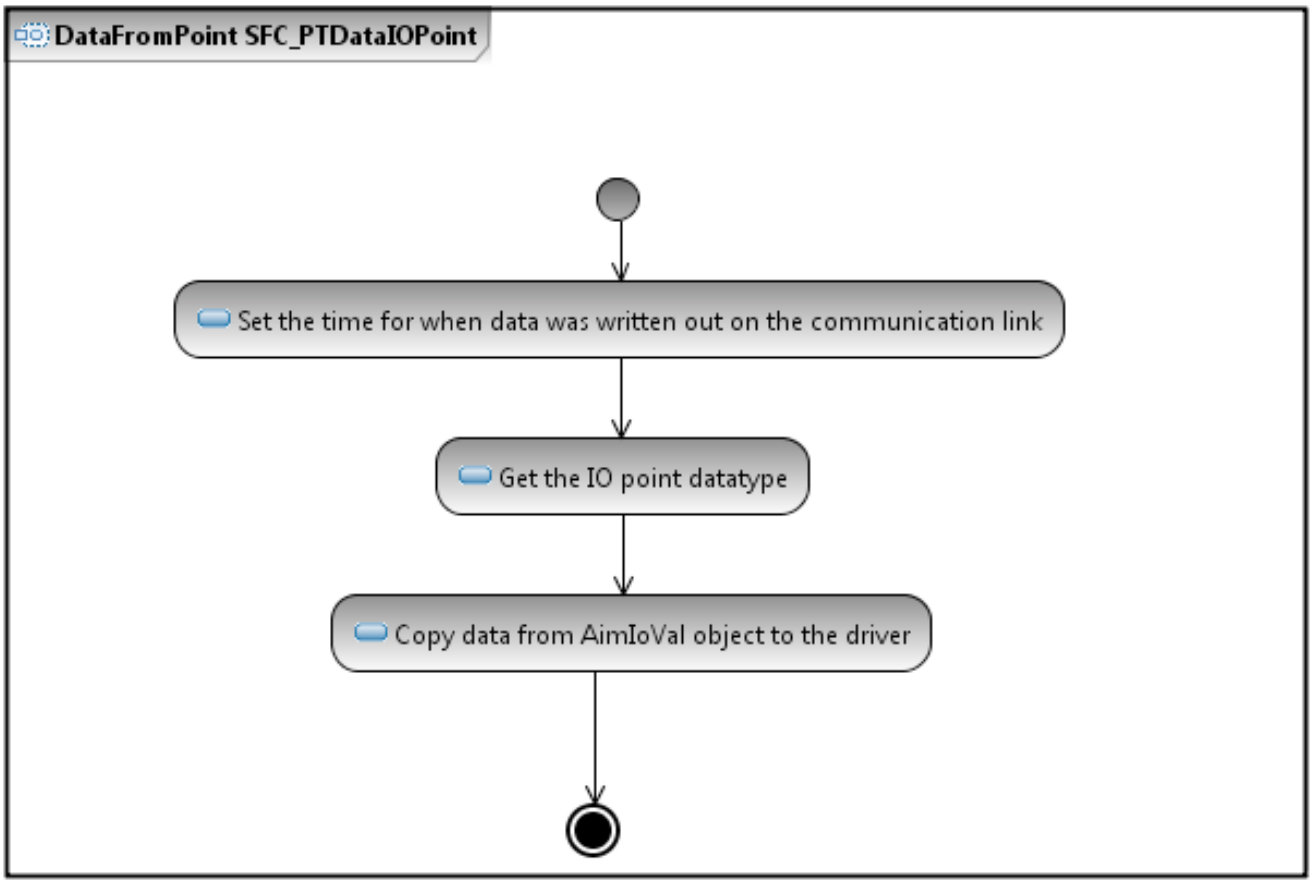

FIGURE 35: SFC DRIVER DATAFROMPOIINT SFC_PTDATAIOPOINT ACTIVITY DIAGRAM.

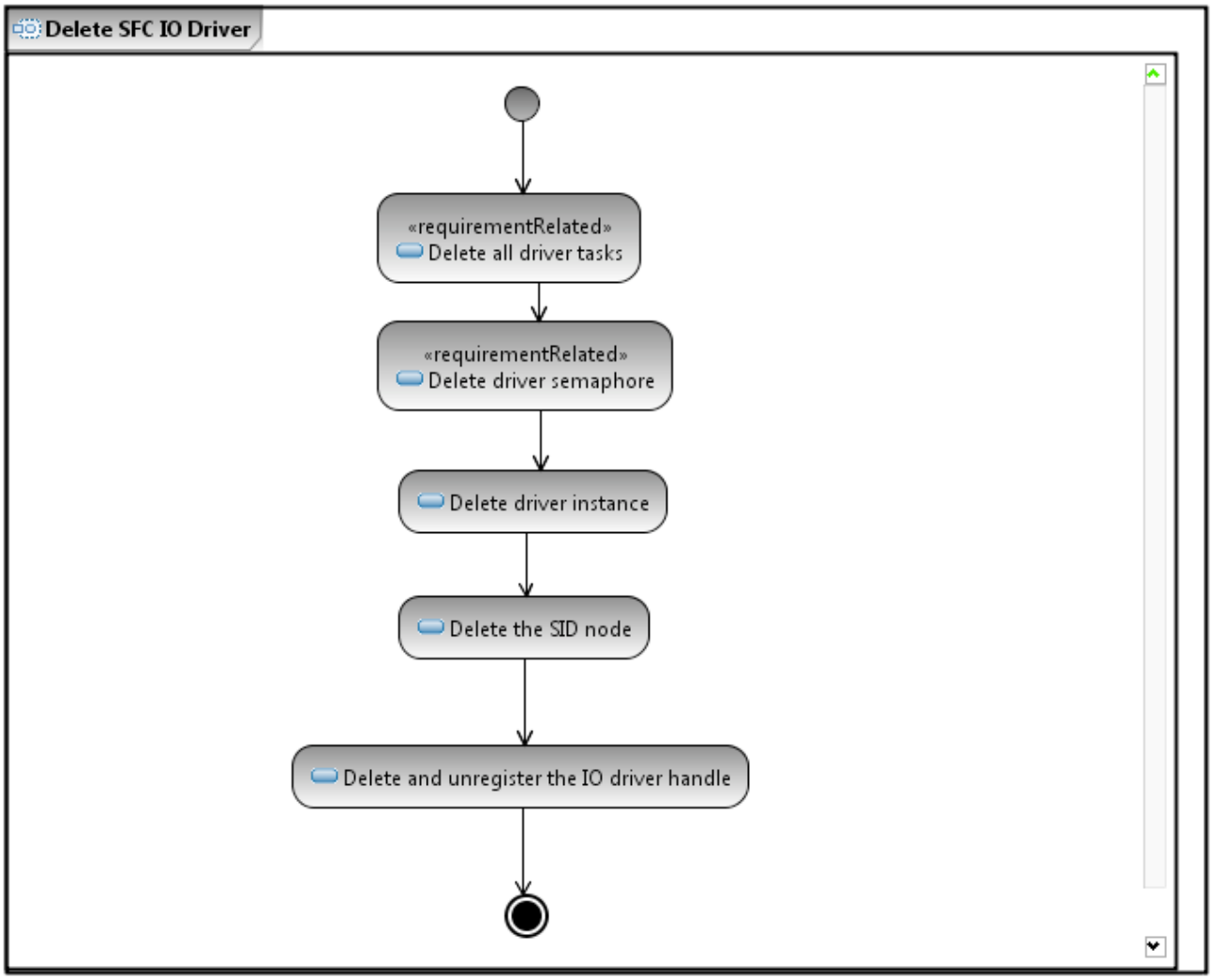

FIGURE 36: SFC DRIVER DELETE SFC IODRIVER ACTIVITY DIAGRAM.

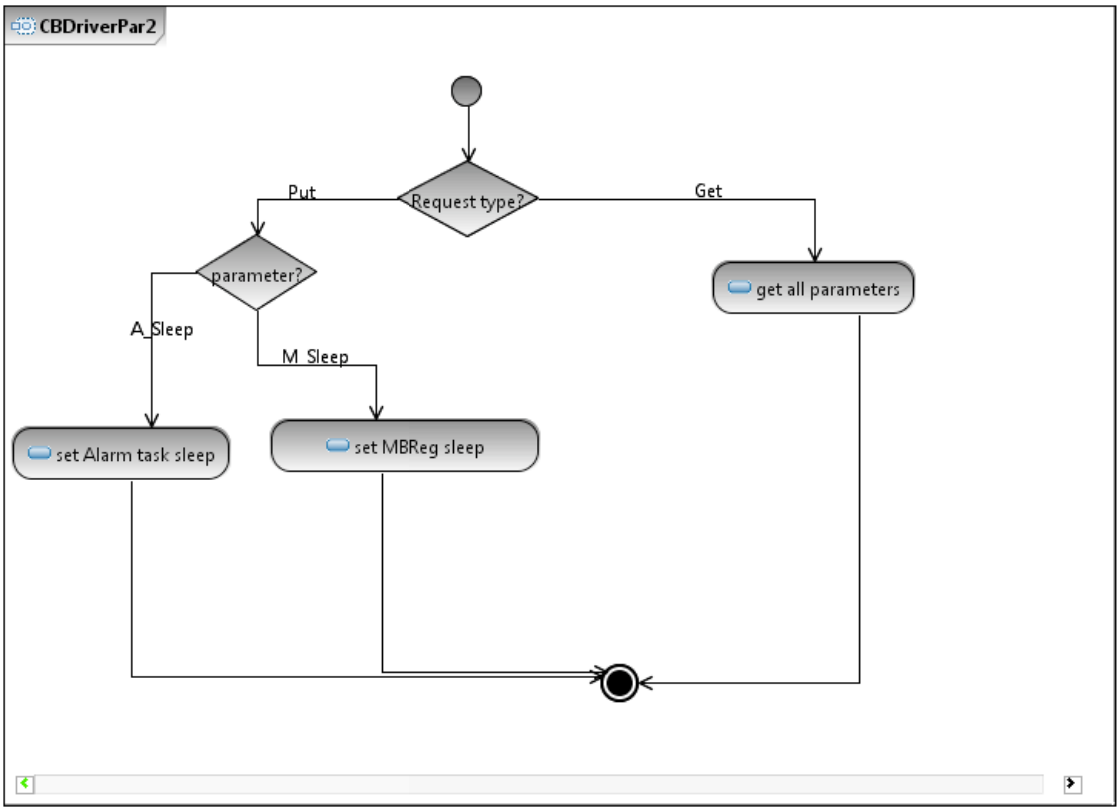

FIGURE 37: SFC DRIVER CBDRIVERPAR2 ACTIVITY DIAGRAM.

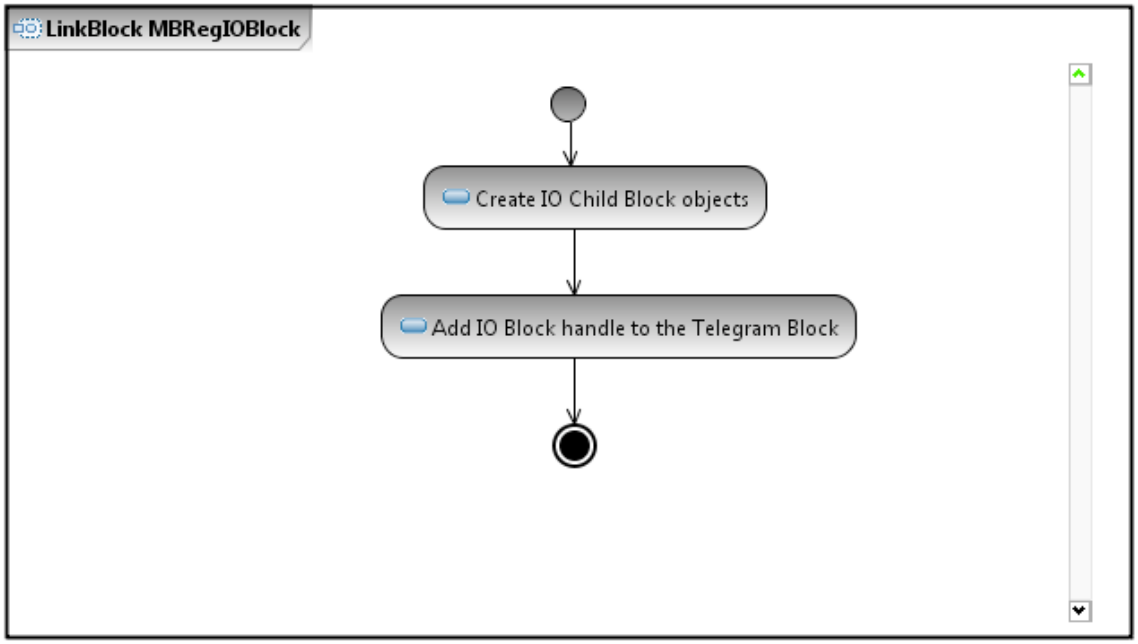

FIGURE 38: SFC DRIVER LINKBLOCK MBREGIOBLOCK ACTIVITY DIAGRAM.

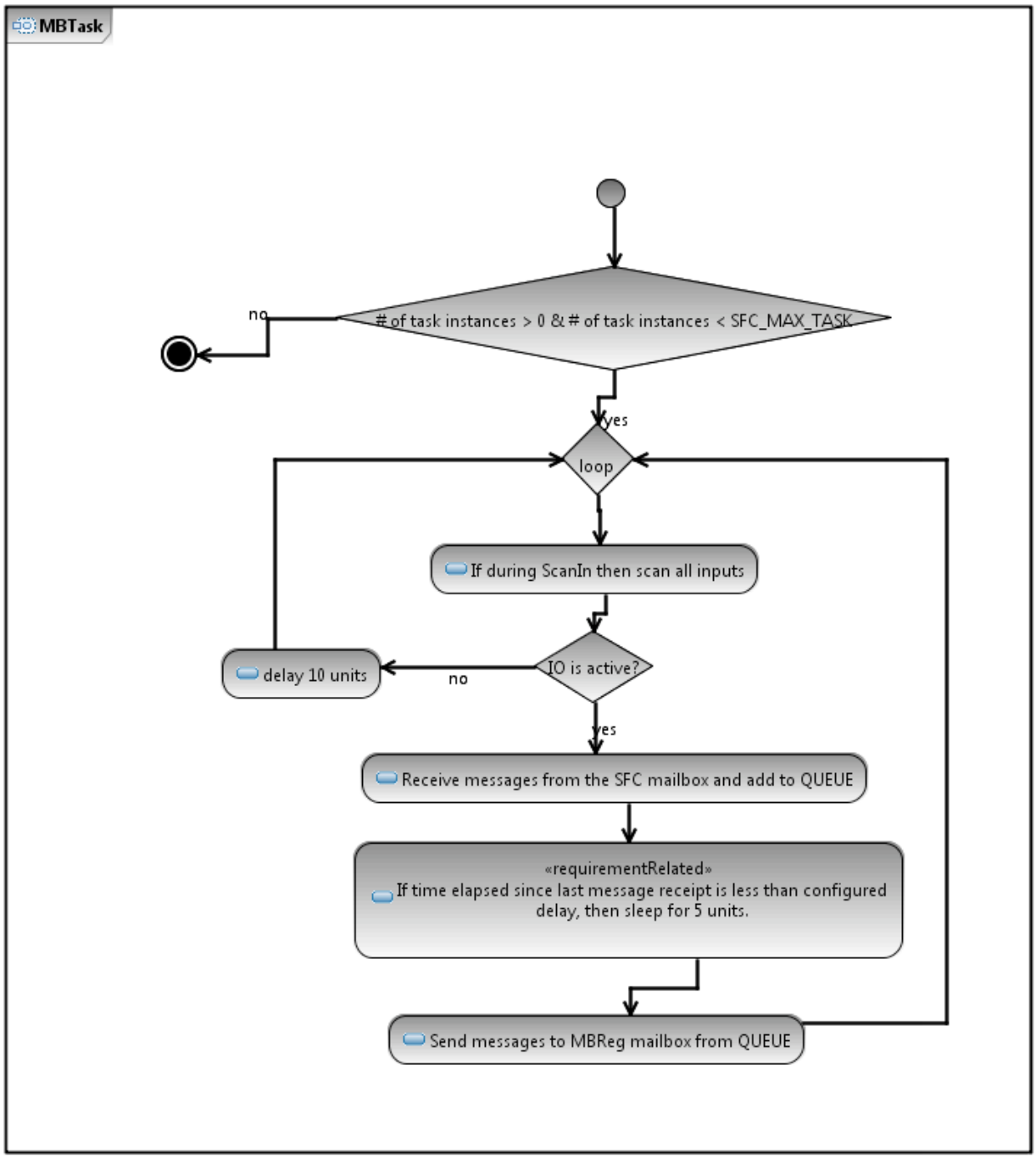

FIGURE 39: SFC DRIVER MBTASK ACTIVITY DIAGRAM.

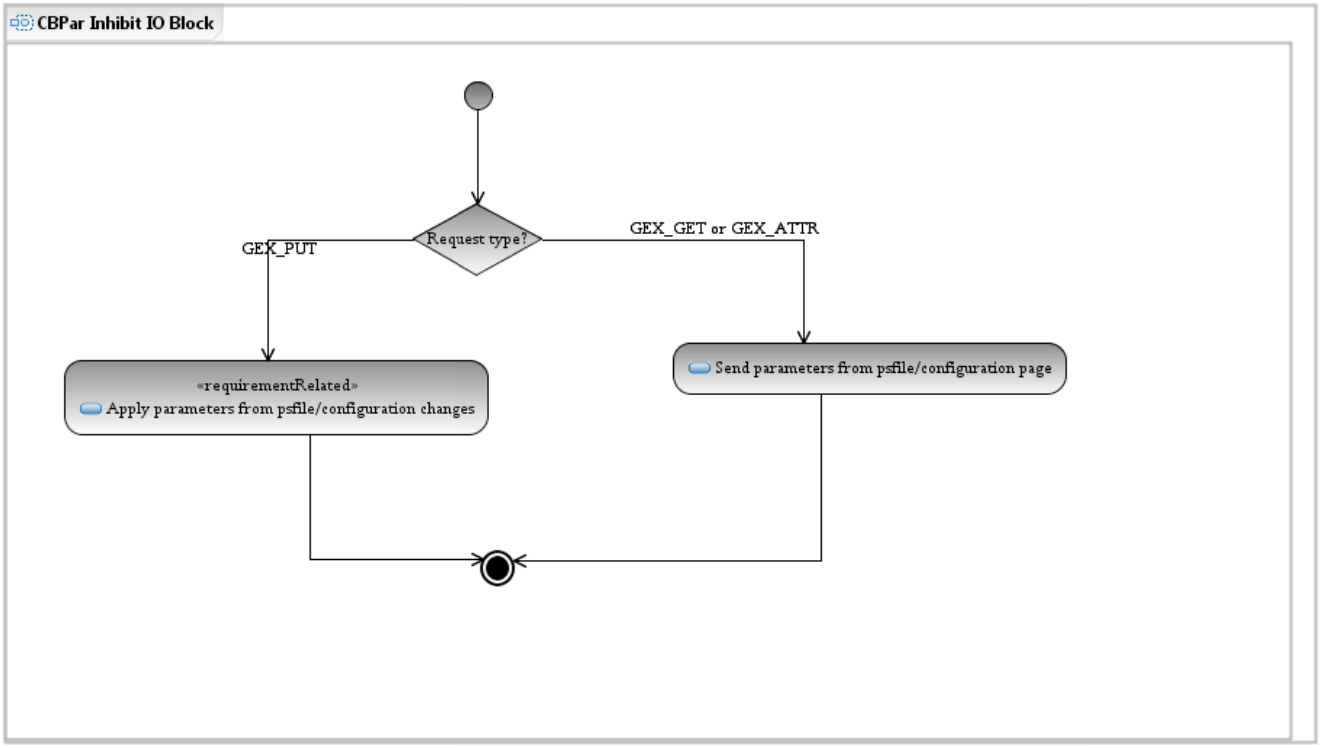

FIGURE 40:SFC DRIVER CBPAR INHIBITIOBLOCK ACTIVITY DIAGRAM.

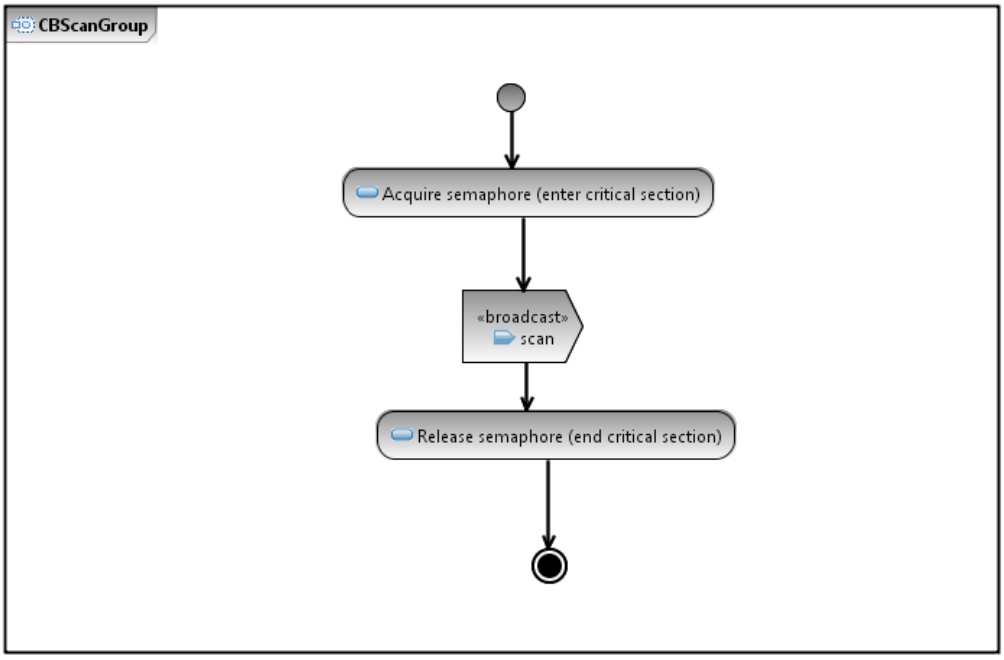

FIGURE 41: SFC DRIVER CBSCANGROUP ACTIVITY DIAGRAM.

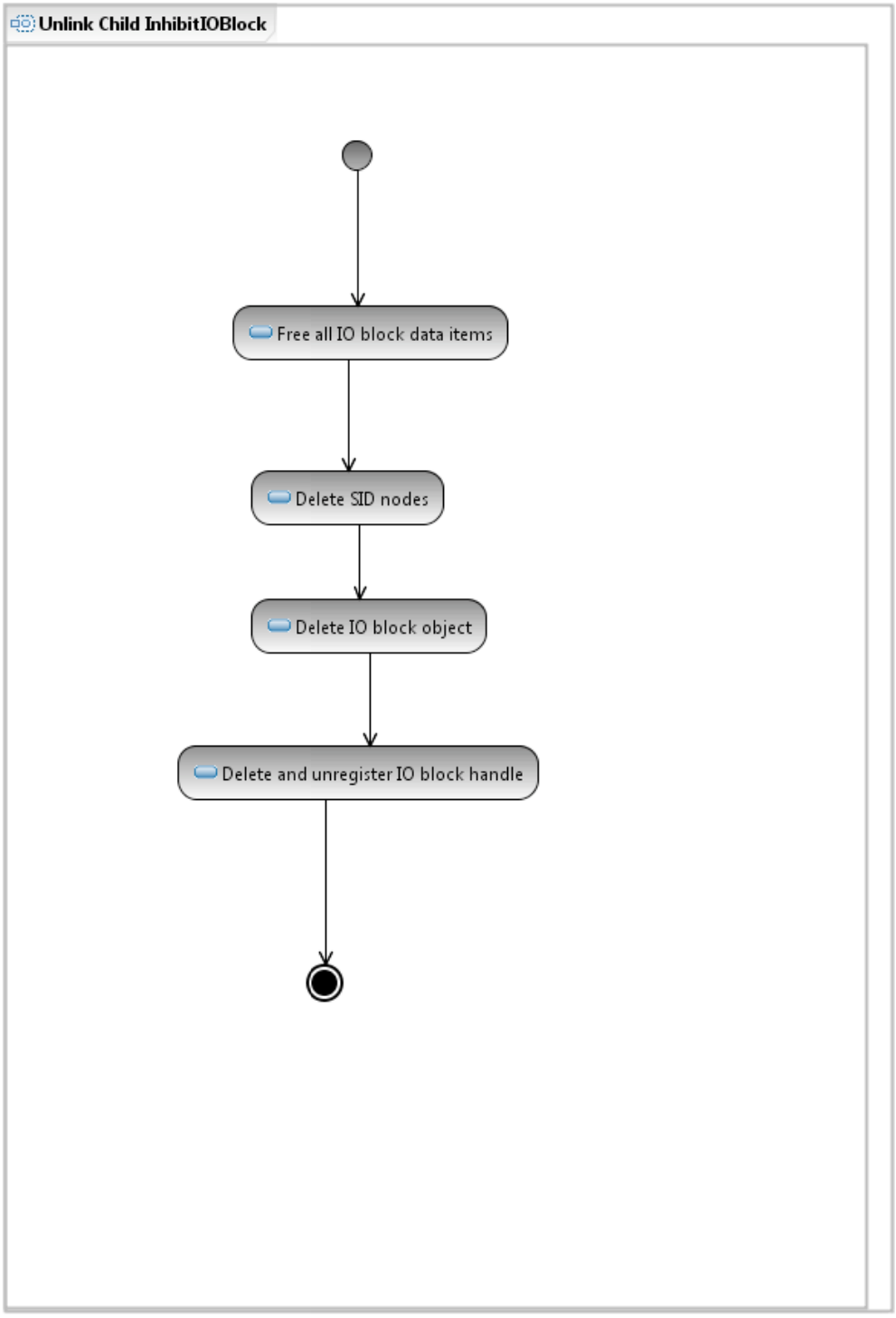

FIGURE 42: SFC DRIVER UNLINK CHILD INHIBITIOBLOCK ACTIVITY DIAGRAM.

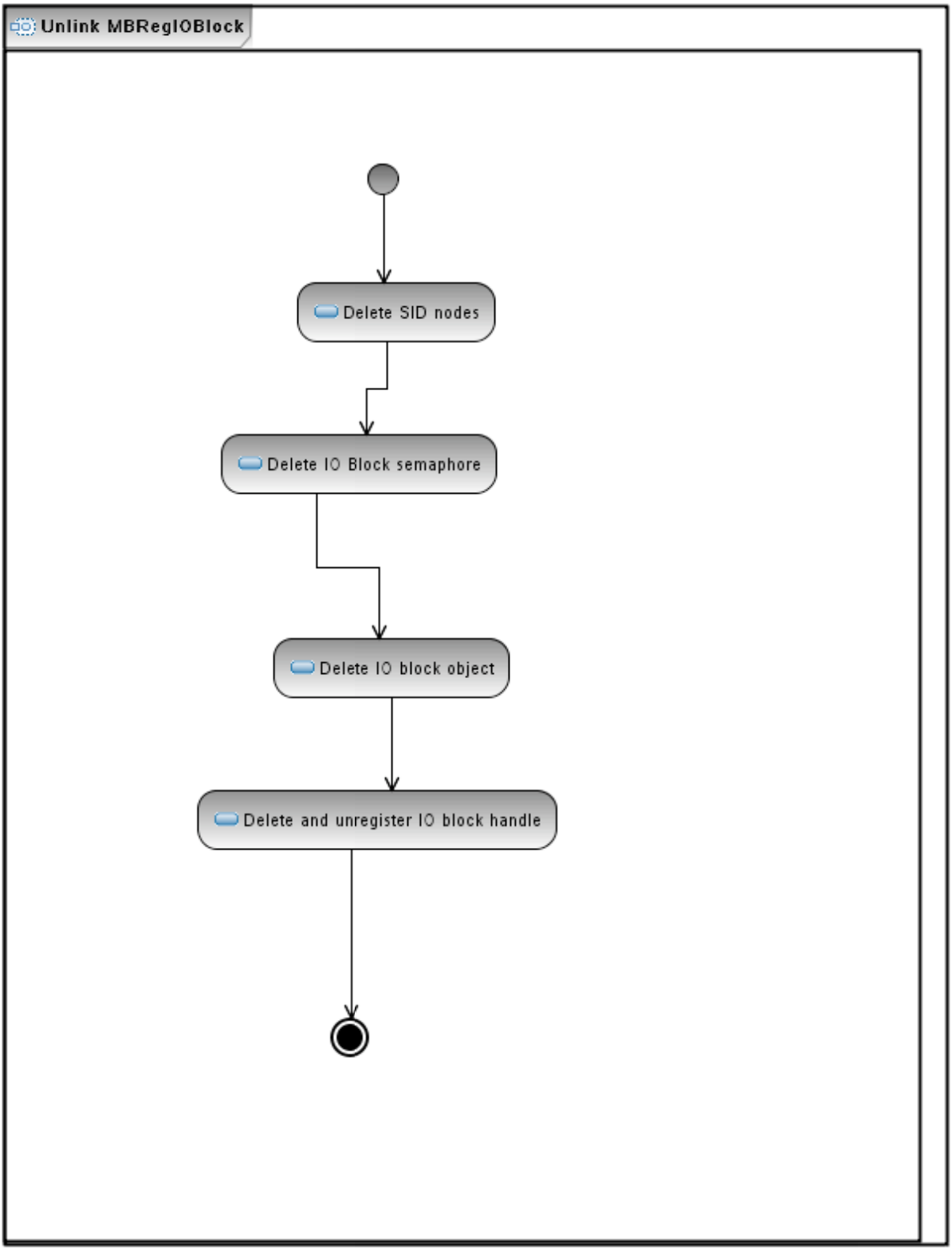

FIGURE 43: SFC DRIVER UNLINK MBREG IOBLOCK ACTICITY DIAGRAM.

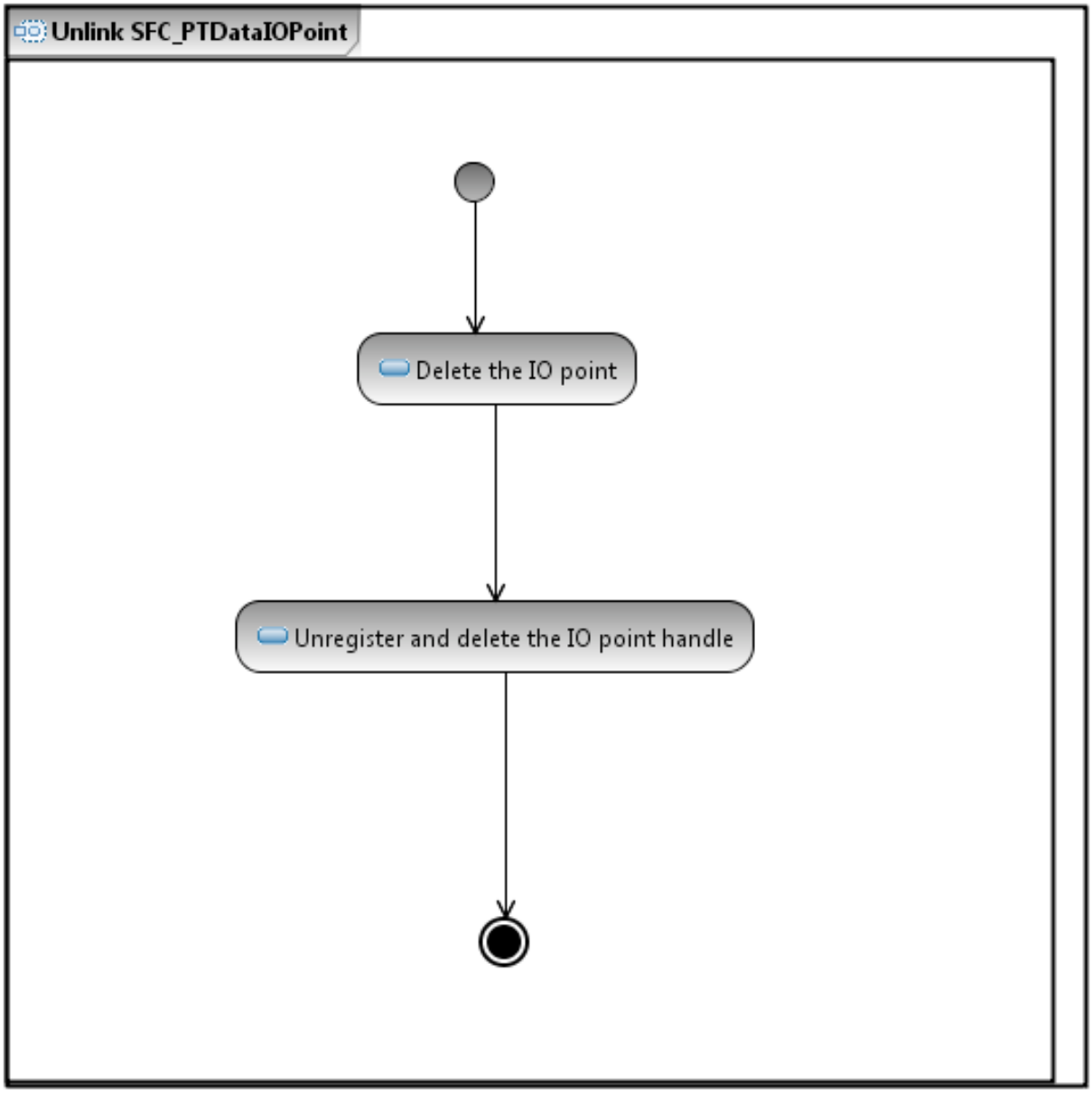

FIGURE 44: SFC DRIVER UNLINK SFC_PTDATAIOPOINT ACTIVITY DIAGRAM.

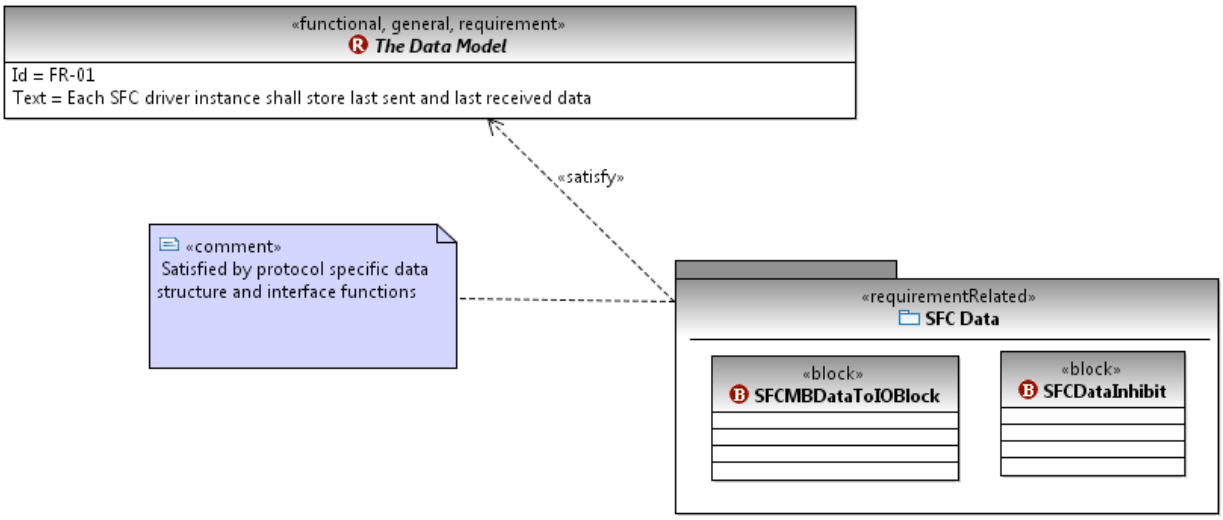

FIGURE 45: SFC DRIVER FUNCTIONAL REQUIREMENT 1.

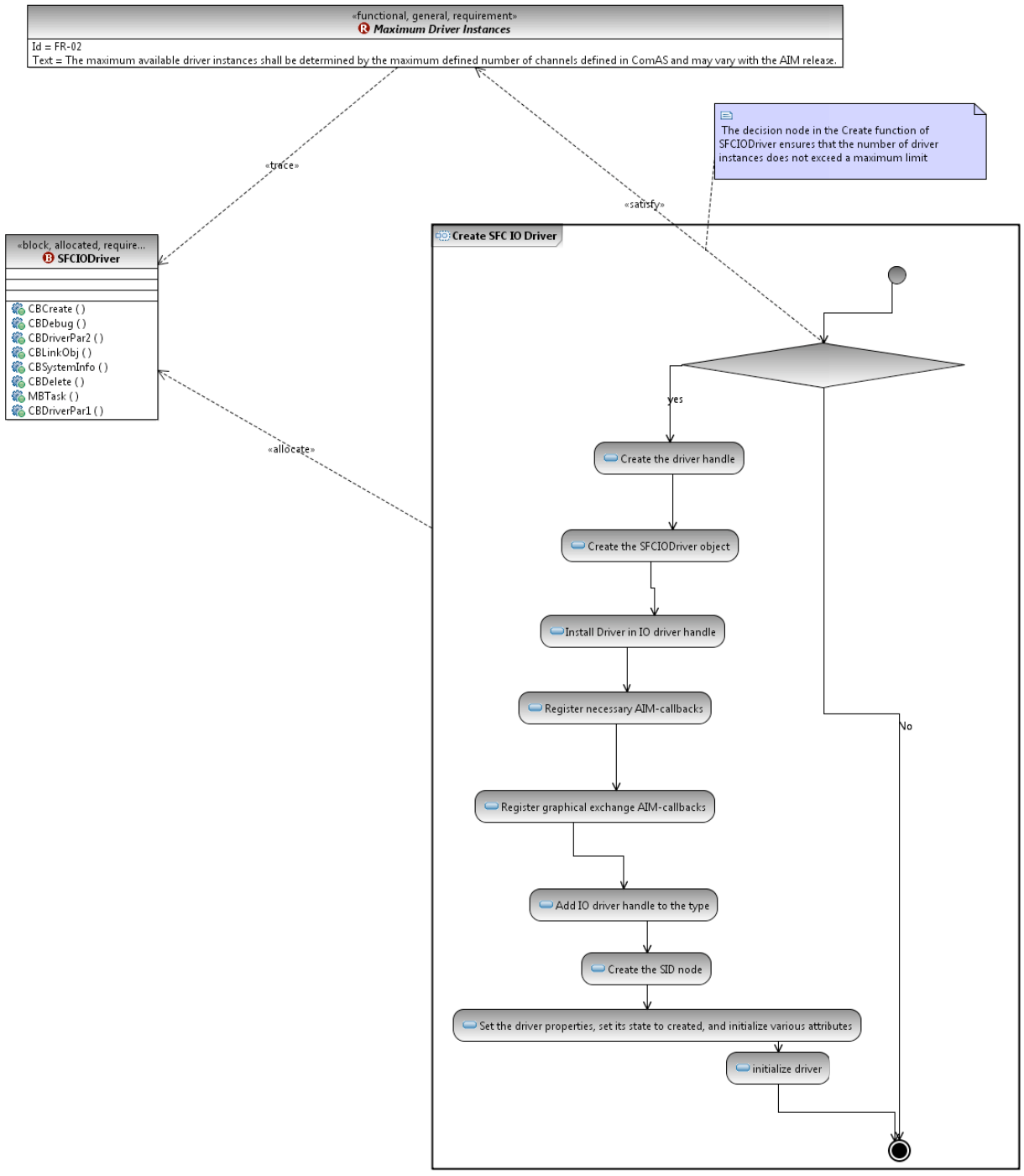

FIGURE 46: SFC DRIVER FUNCTIONAL REQUIREMENT 2.

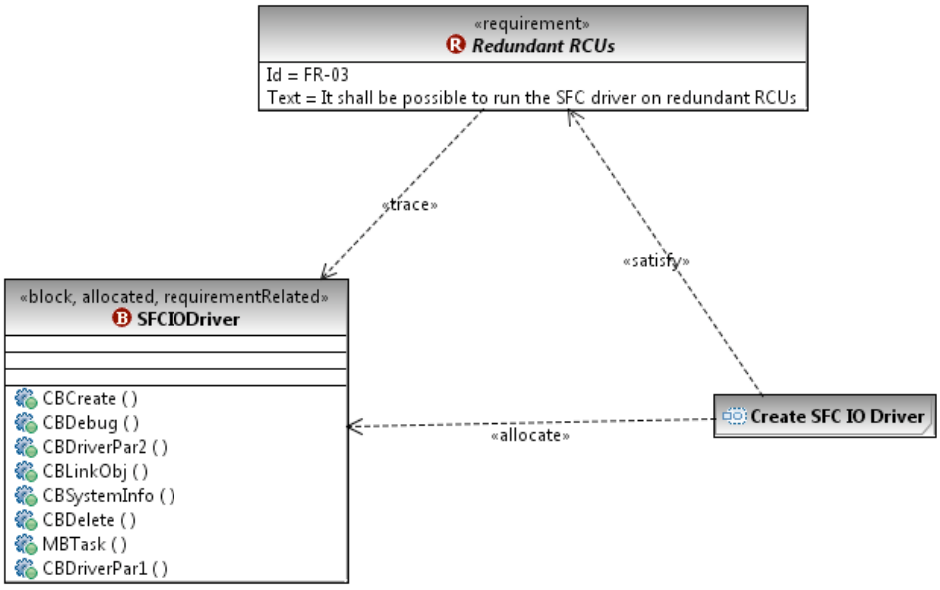

FIGURE 47: SFC DRIVER FUNCTIONAL REQUIREMENT 3.

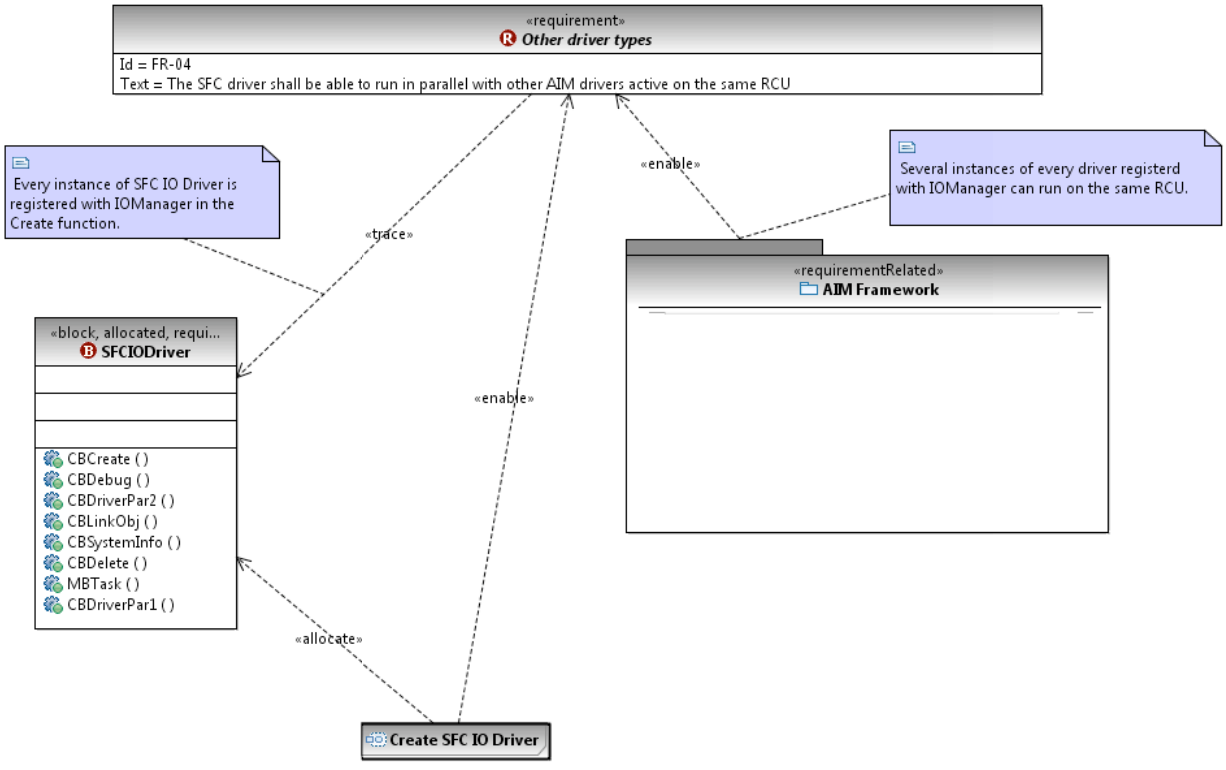

FIGURE 48: SFC DRIVER FUNCTIONAL REQUIREMENT 4.

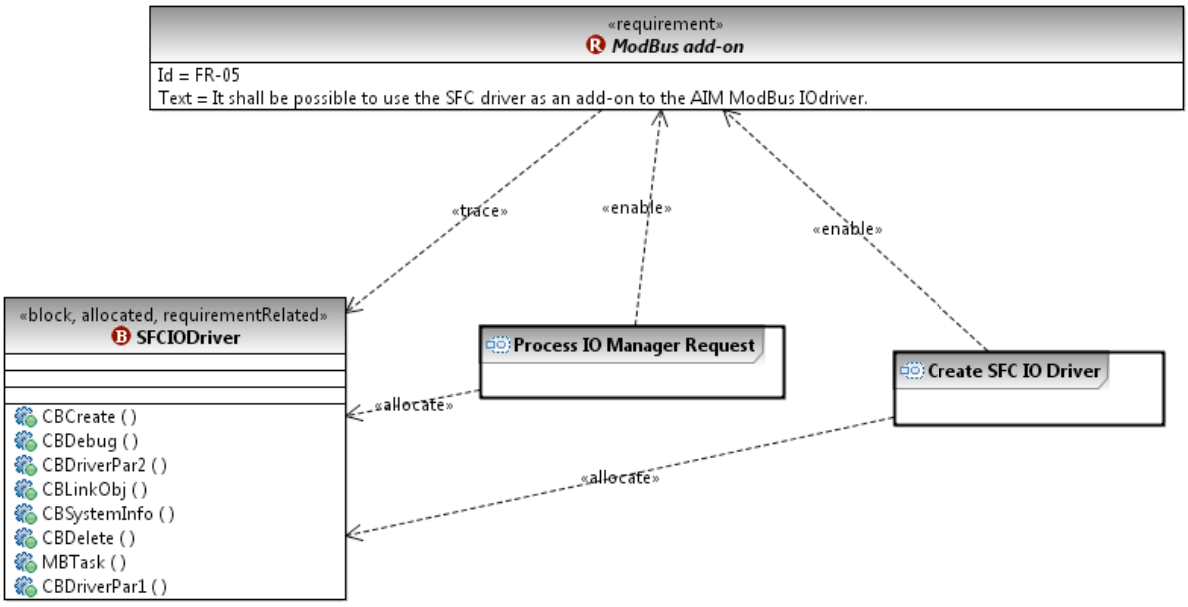

FIGURE 49: SFC DRIVER FUNCTIONAL REQUIREMENT 5.

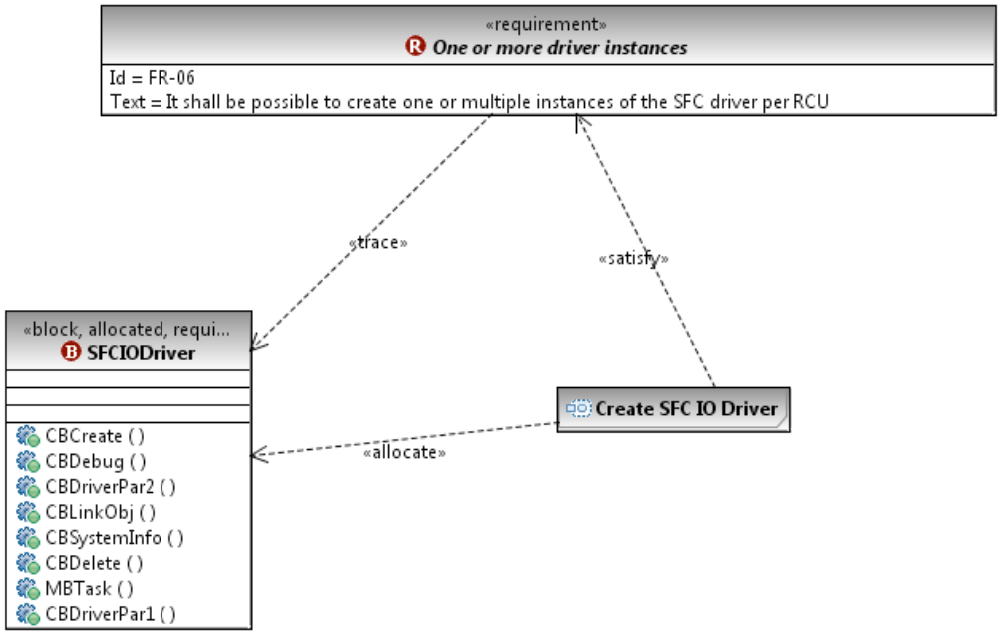

FIGURE 50: SFC DRIVER FUNCTIONAL REQUIREMENT 6.

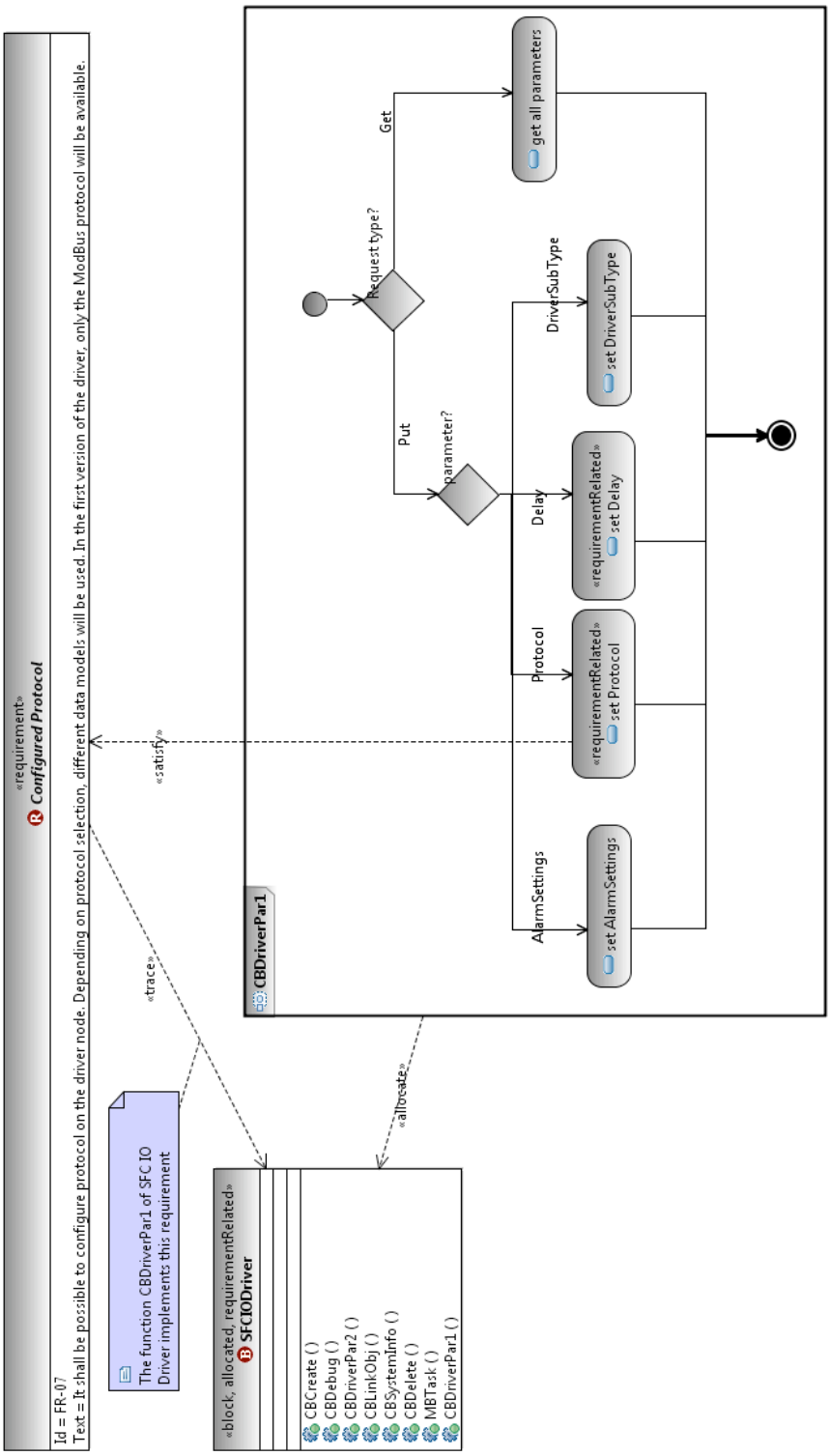

FIGURE 51: SFC DRIVER FUNCTIONAL REQUIREMENT 7.

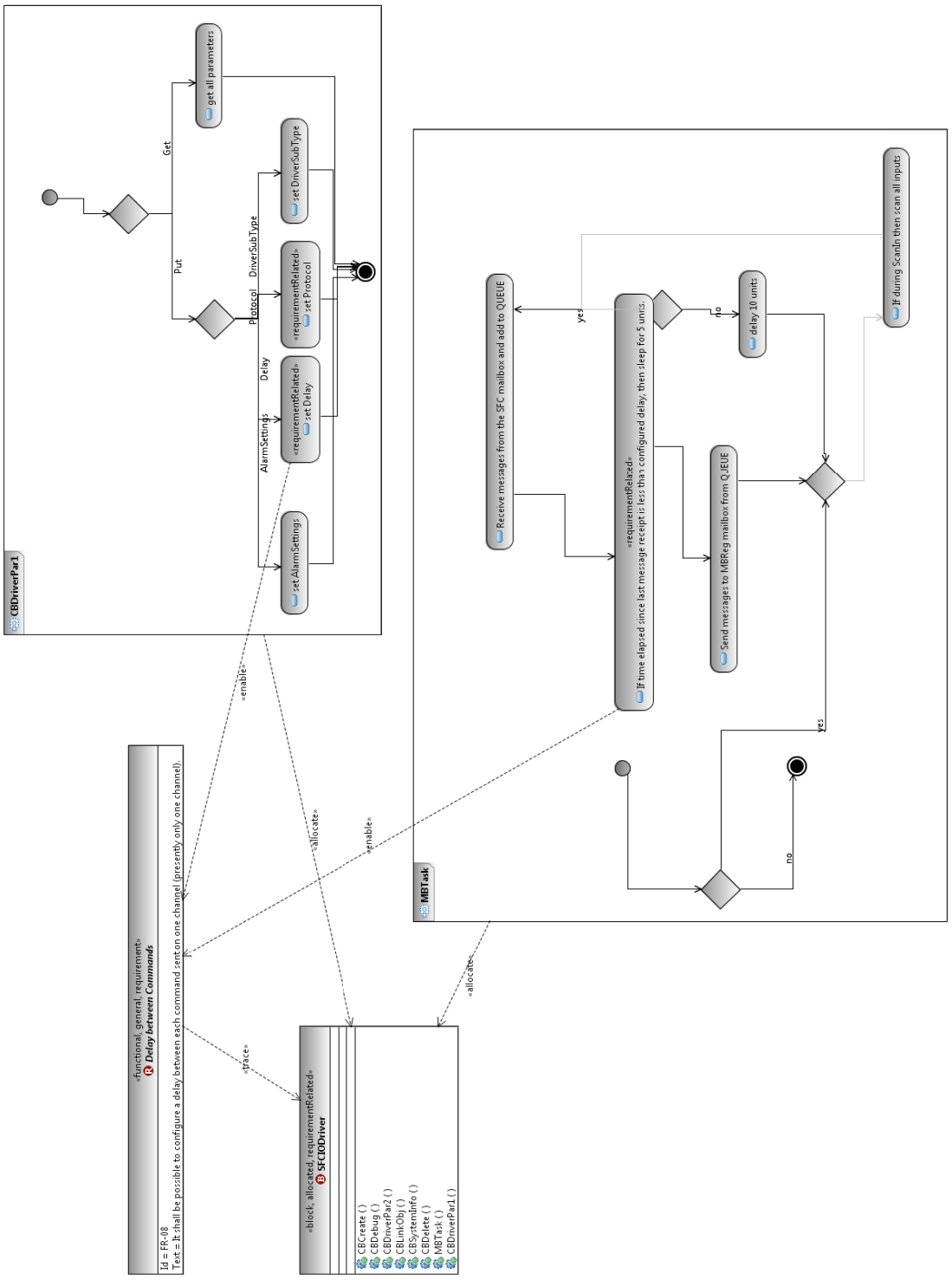

FIGURE 52: SFC DRIVER FUNCTIONAL REQUIREMENT 8.

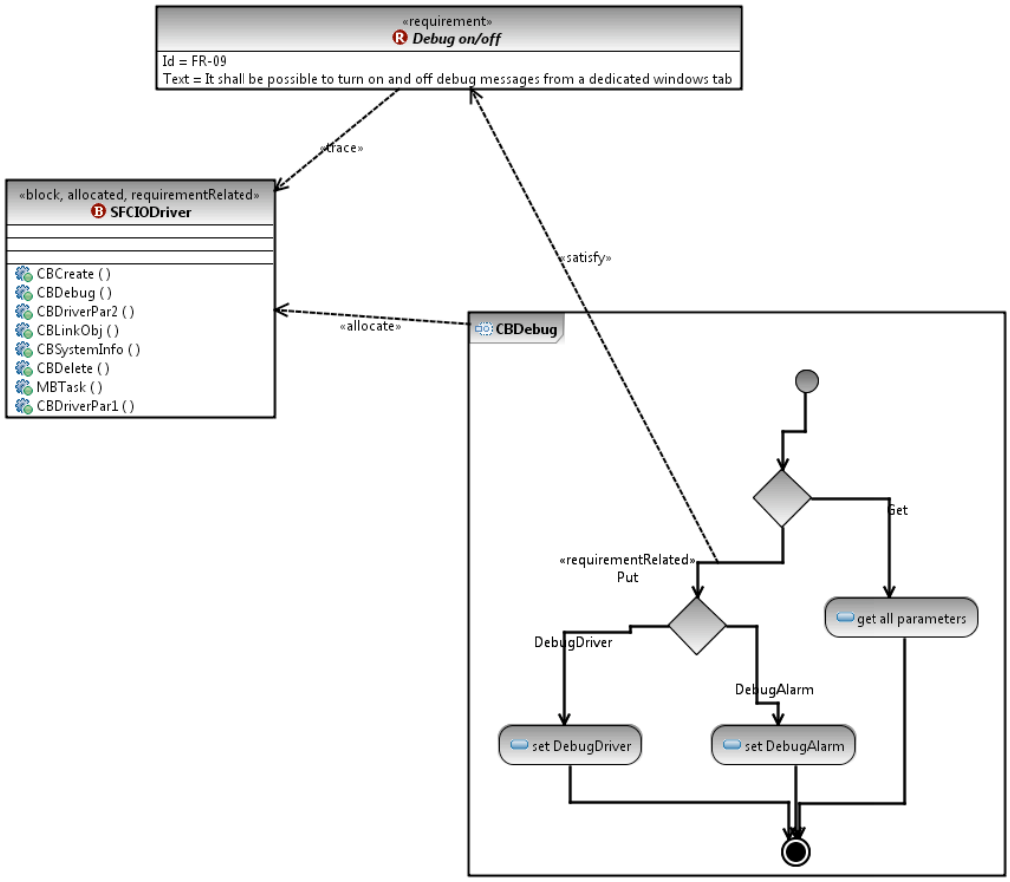

FIGURE 53: SFC DRIVER FUNCTIONAL REQUIREMENT 9.

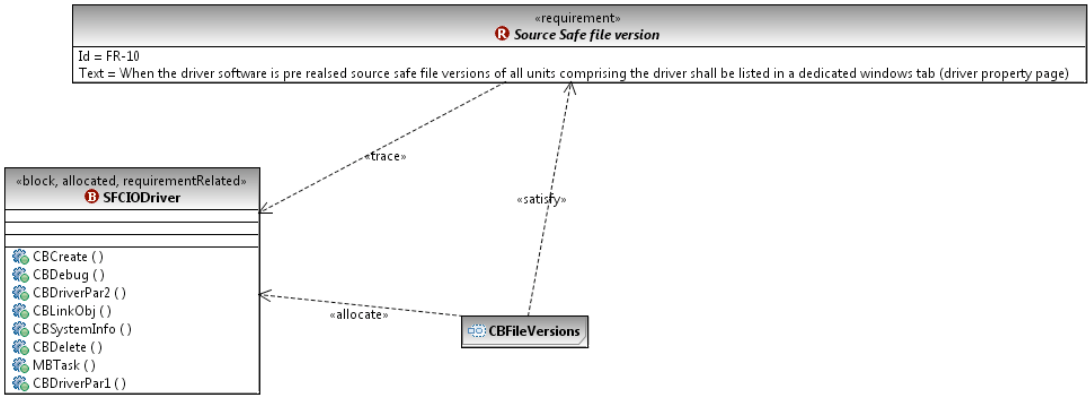

FIGURE 54: SFC DRIVER FUNCTIONAL REQUIREMENT 10.

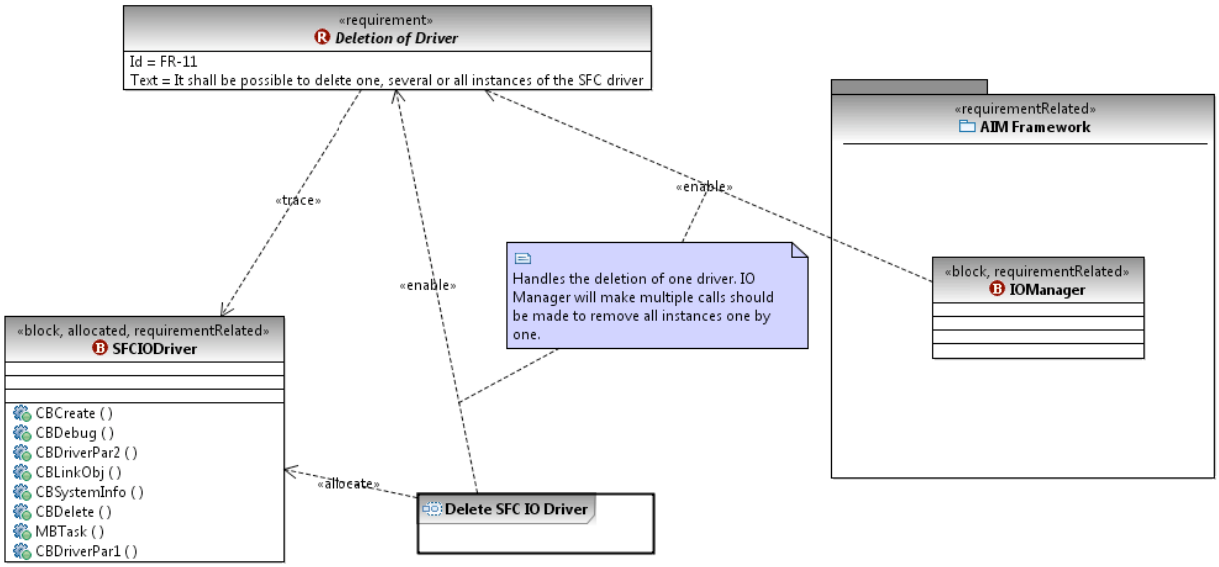

FIGURE 55: SFC DRIVER FUNCTIONAL REQUIREMENT 11.

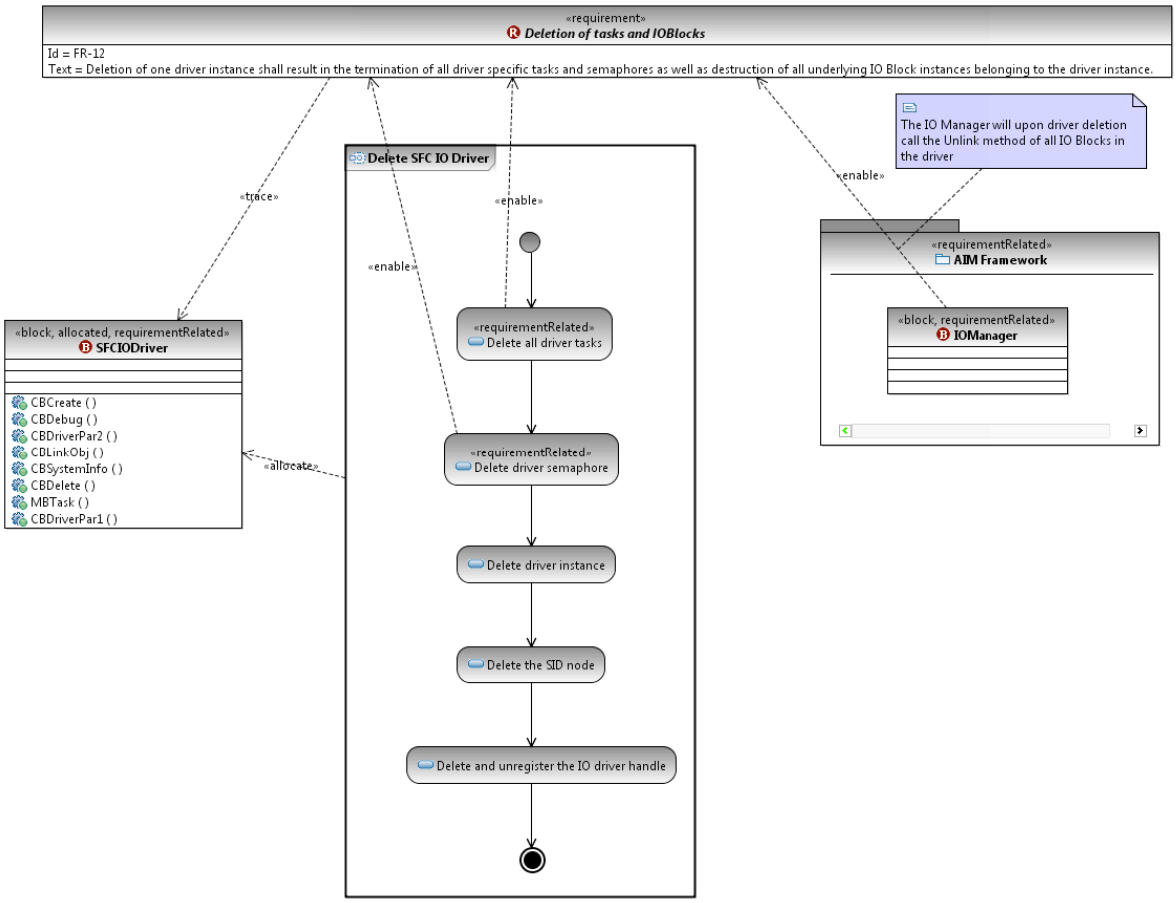

FIGURE 56: SFC DRIVER FUNCTIONAL REQUIREMENT 12.

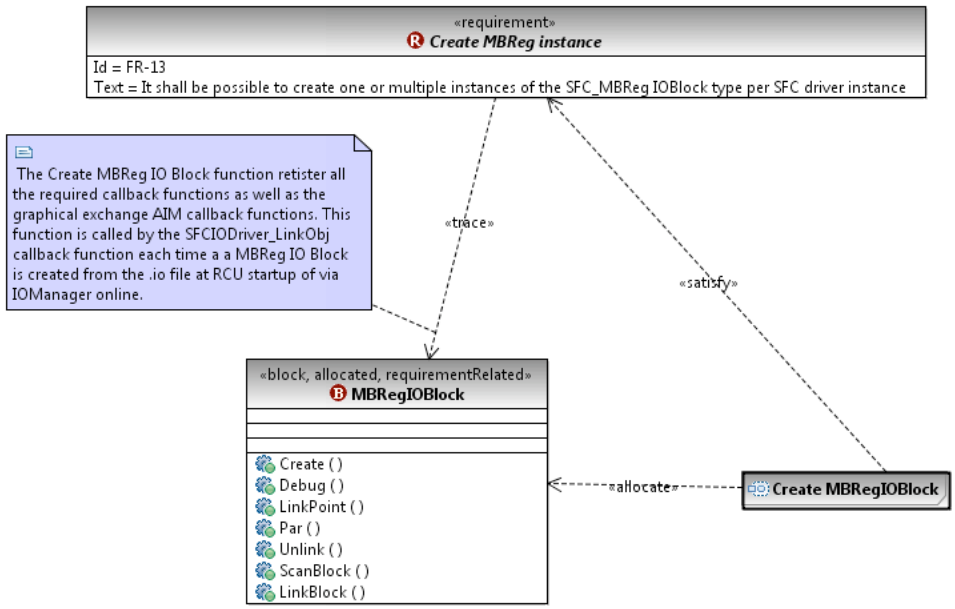

FIGURE 57: SFC DRIVER FUNCTIONAL REQUIREMENT 13.

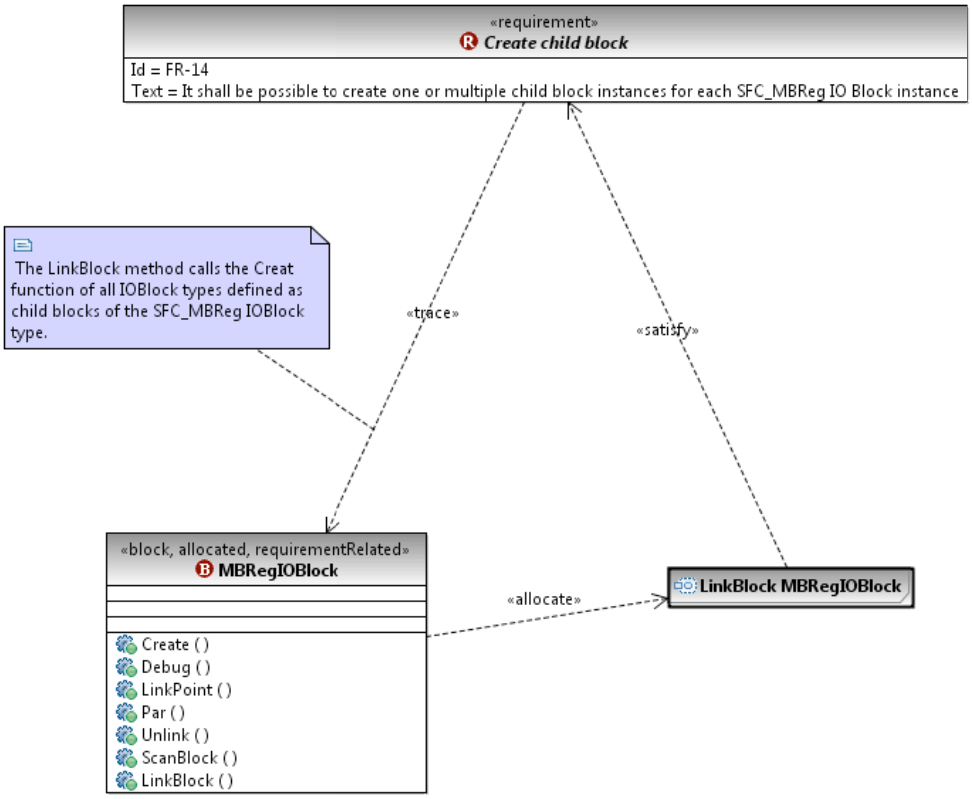

FIGURE 58: SFC DRIVER FUNCTIONAL REQUIREMENT 14.

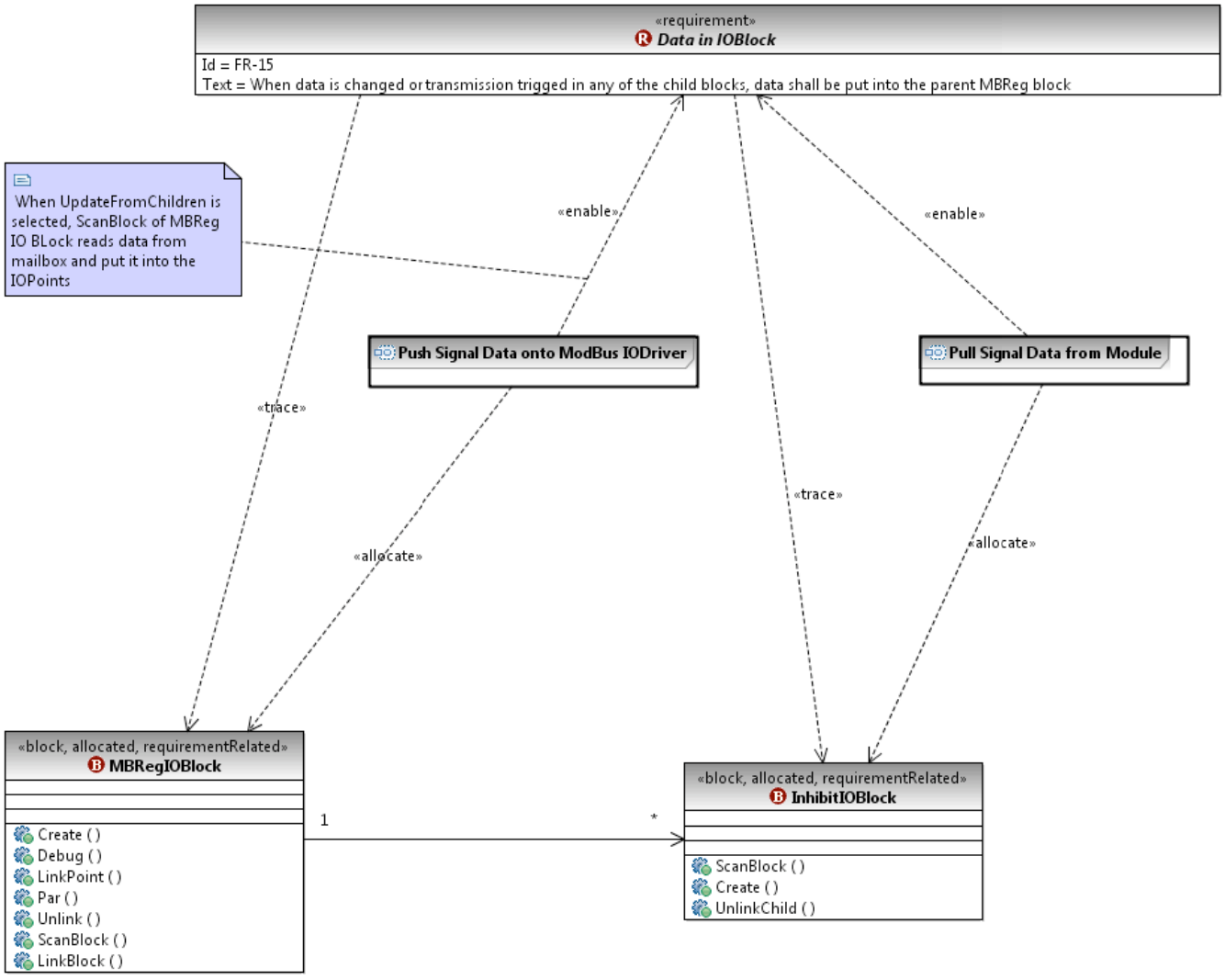

FIGURE 59: SFC DRIVER FUNCTIONAL REQUIREMENT 15.

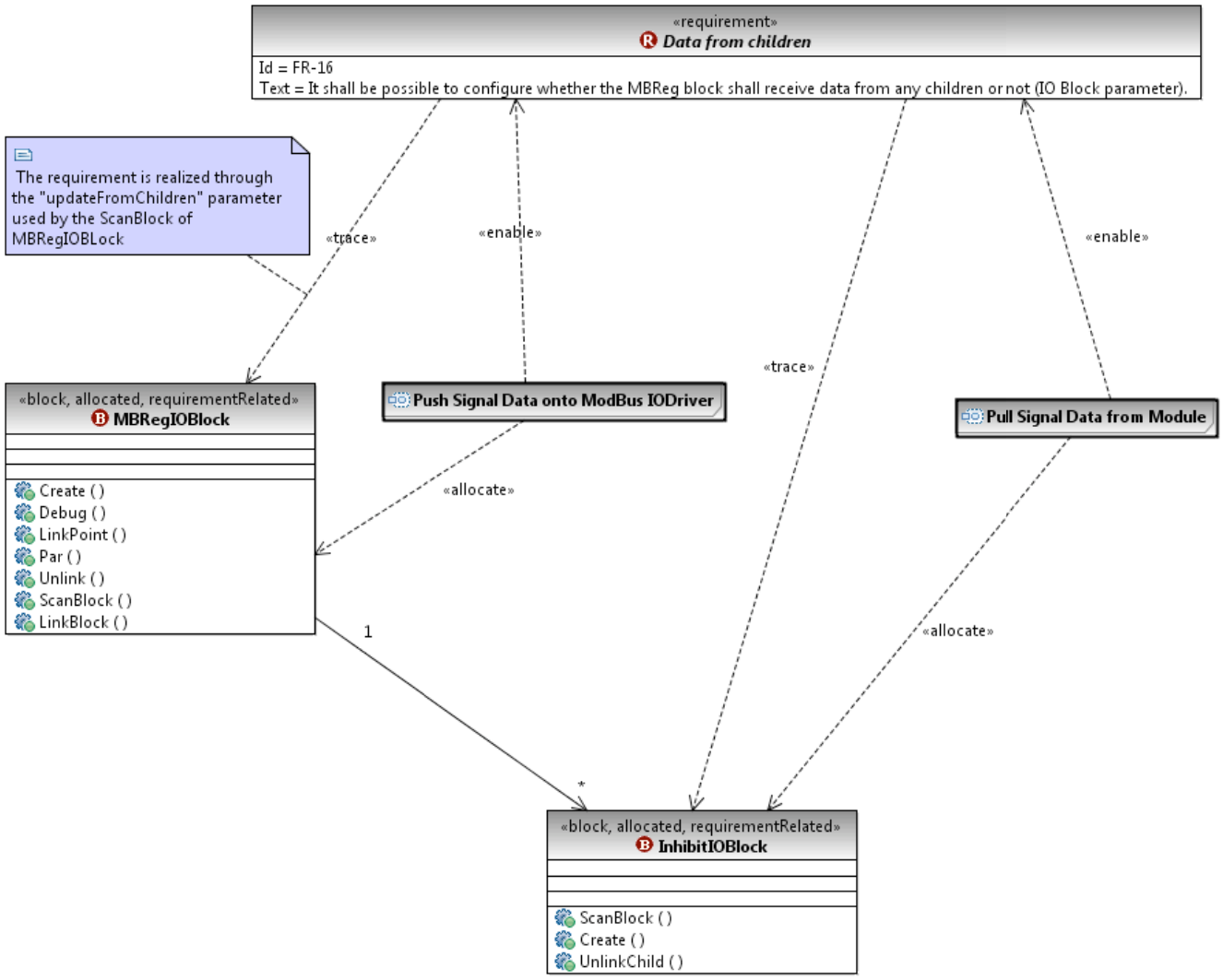

FIGURE 60: SFC DRIVER FUNCTIONAL REQUIREMENT 16.

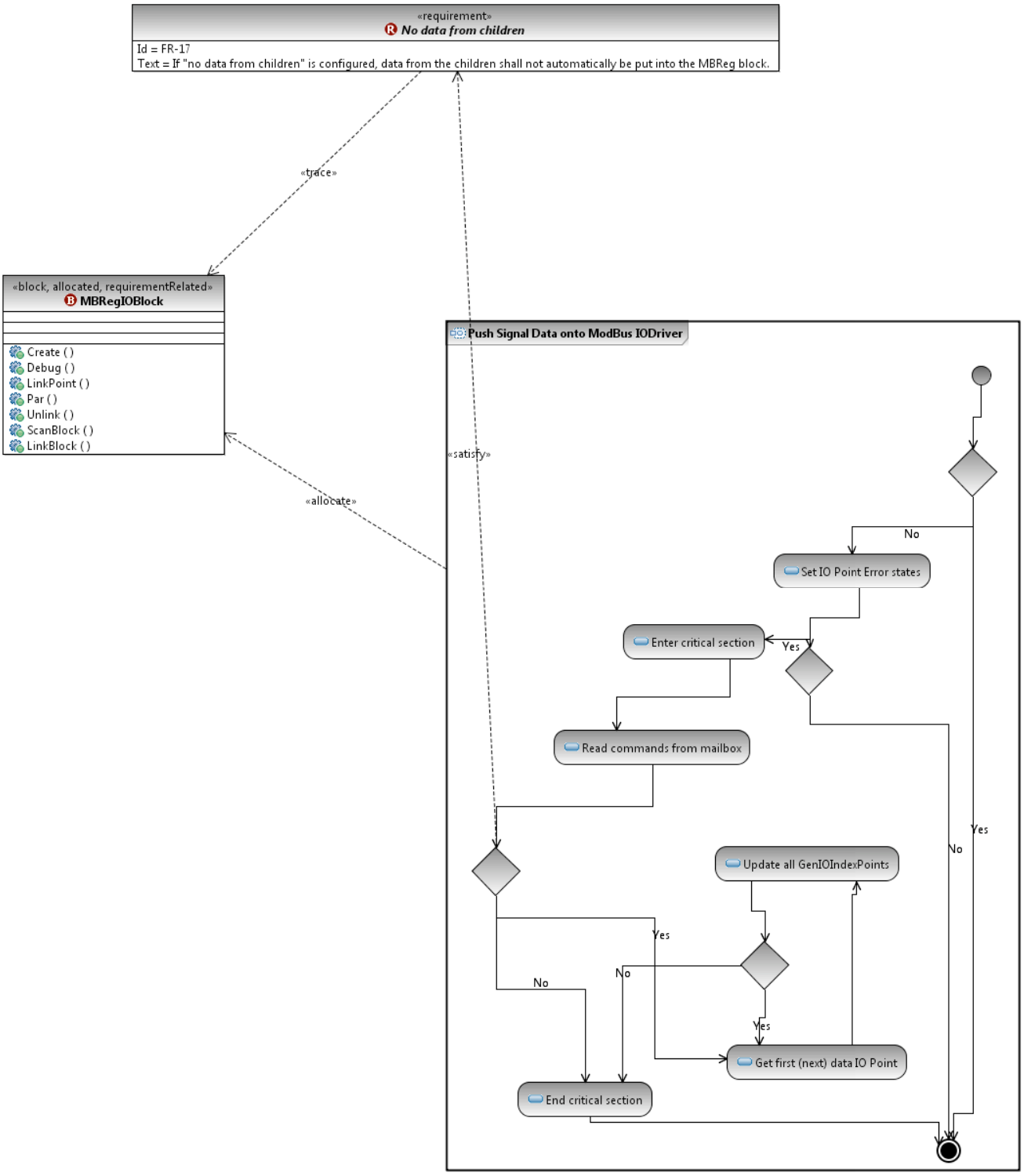

FIGURE 61: SFC DRIVER FUNCTIONAL REQUIREMENT 17.

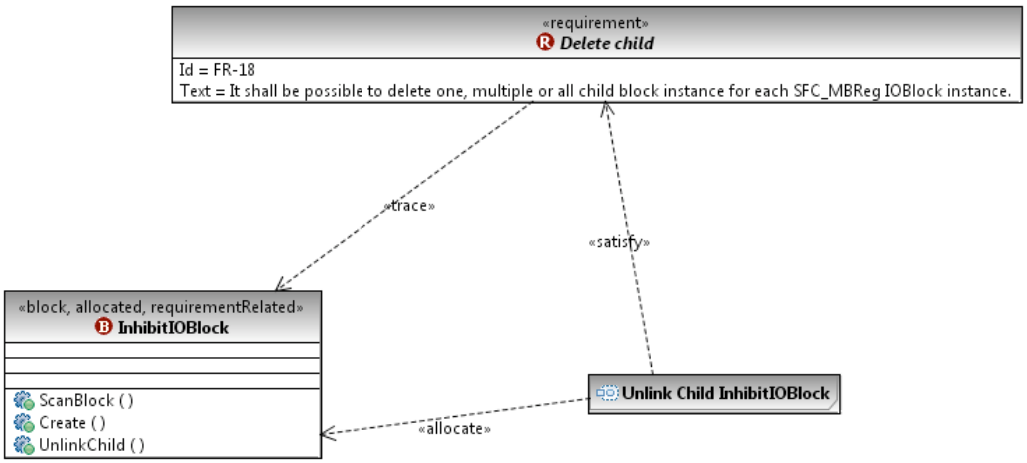

FIGURE 62: SFC DRIVER FUNCTIONAL REQUIREMENT 18.

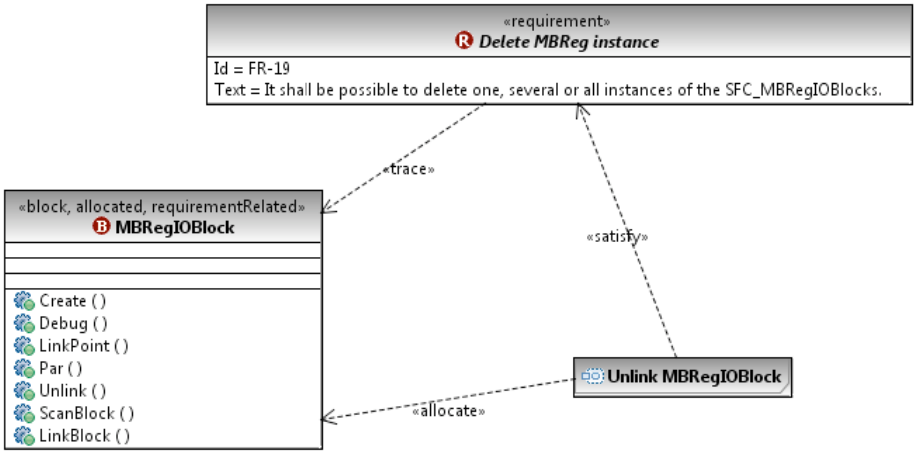

FIGURE 63: SFC DRIVER FUNCTIONAL REQUIREMENT 19.

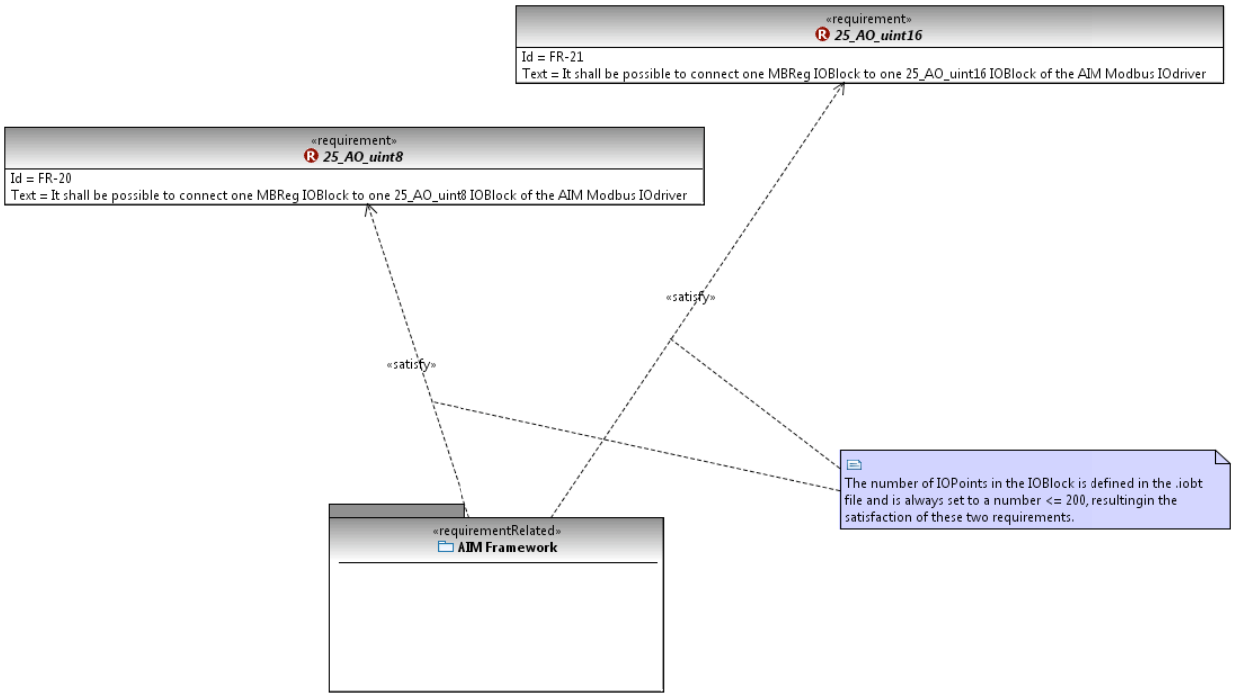

FIGURE 64: SFC DRIVER FUNCTIONAL REQUIREMENTS 20 & 21

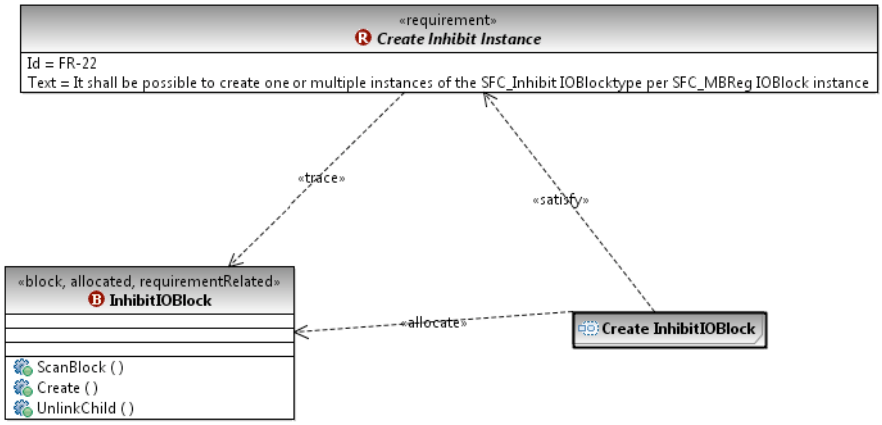

FIGURE 65: SFC DRIVER FUNCTIONAL REQUIREMENT 22.

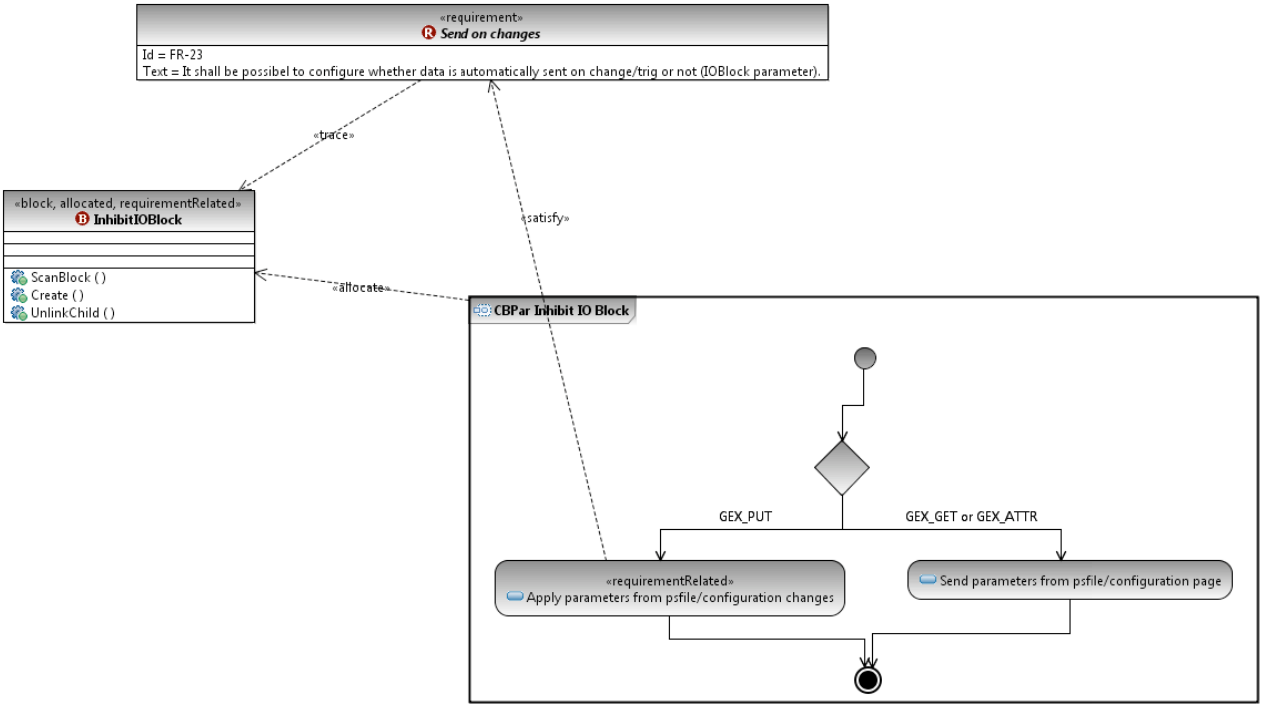

FIGURE 66: SFC DRIVER FUNCTIONAL REQUIREMENT 23.

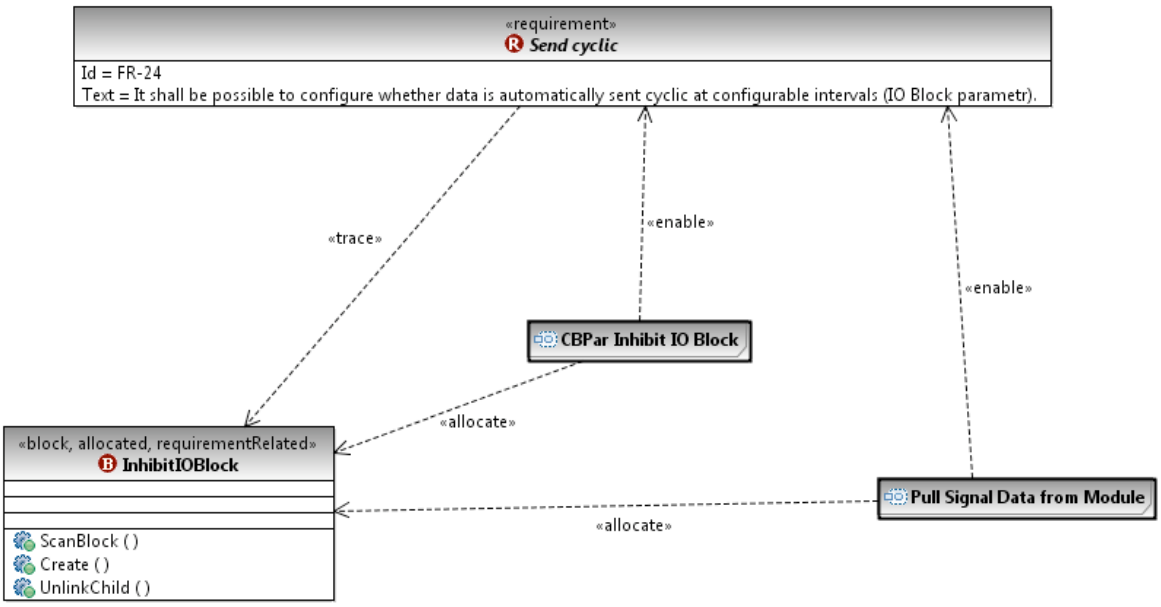

FIGURE 67: SFC DRIVER FUNCTIONAL REQUIREMENT 24.

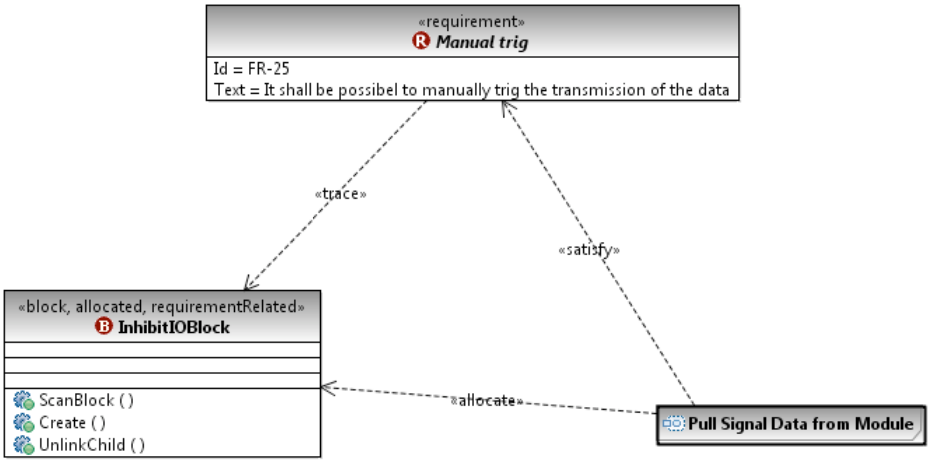

FIGURE 68: SFC DRIVER FUNCTIONAL REQUIREMENT 25.

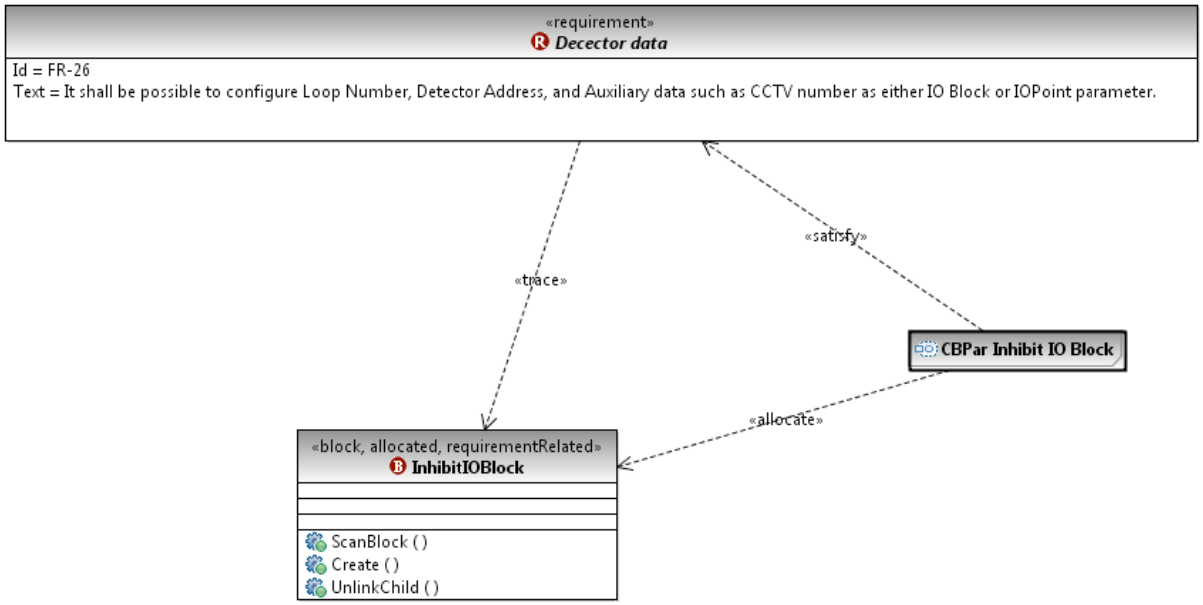

FIGURE 69: SFC DRIVER FUNCTIONAL REQUIREMENT 26.

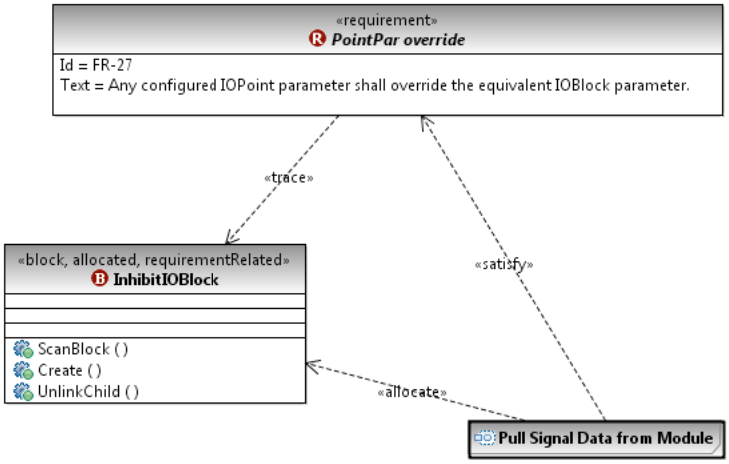

FIGURE 70: SFC DRIVER FUNCTIONAL REQUIREMENT 27.

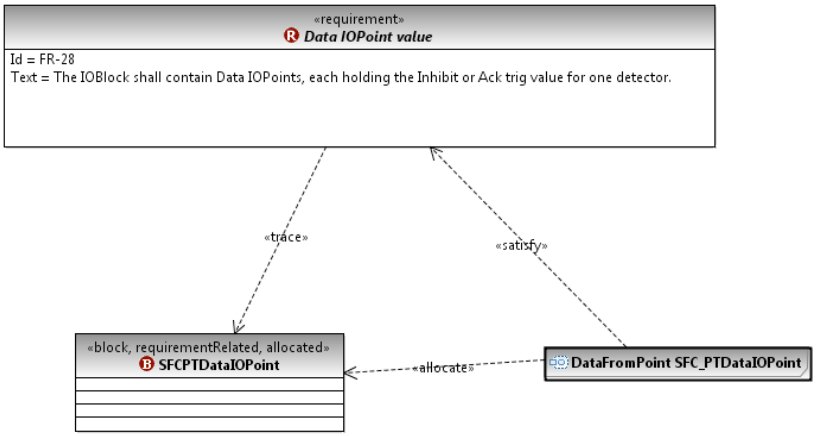

FIGURE 71: SFC DRIVER FUNCTIONAL REQUIREMENT 28.

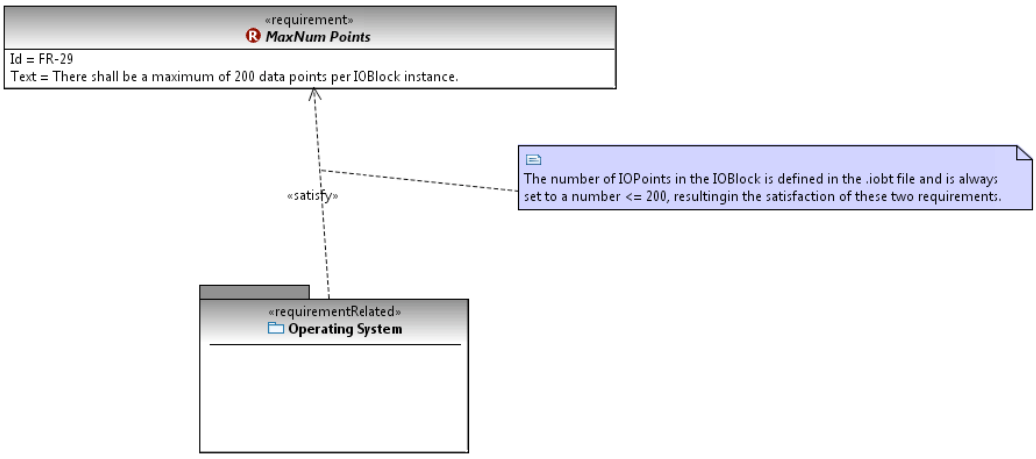

FIGURE 72: SFC DRIVER FUNCTIONAL REQUIREMENT 29.

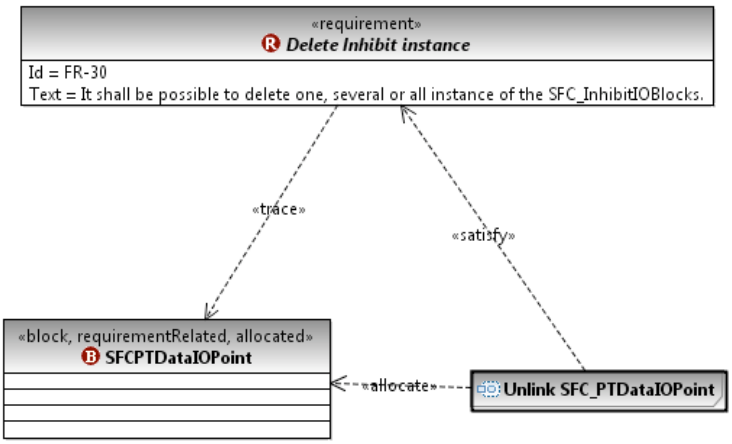

FIGURE 73: SFC DRIVER FUNCTIONAL REQUIREMENT 30.