Using SysML for Modeling of Safety-Critical Software–Hardware Interfaces: Guidelines and Industry Experience

Mehrdad Sabetzadeh[1] Shiva Nejati[1] Lionel Briand[1] Anne-Heidi Evensen Mills[2]

[1]*Certus Software V&V Center, Simula Research Laboratory, Norway*
[2]*Kongsberg Maritime, Kongsberg, Norway*
Email: {*mehrdad,shiva,briand*}*@simula.no* *anne-heidi.evensen.mills@kongsberg.com*

***Abstract*—Safety-critical embedded systems often need to undergo a rigorous certification process to ensure that the safety risks associated with the use of the systems are adequately mitigated. Interfaces between software and hardware components (SW/HW interfaces) play a fundamental role in these systems by linking the systems' control software to either the physical hardware components or to a hardware abstraction layer. Subsequently, safety certification of embedded systems necessarily has to cover the SW/HW interfaces used in these systems. In this paper, we describe a Model Driven Engineering (MDE) approach based on the SysML language, targeted at facilitating the certification of SW/HW interfaces in embedded systems. Our work draws on our experience with maritime and energy systems, but the work should also apply to a broader set of domains, e.g., the automotive sector, where similar design principles are used for (SW/HW) interface design. Our approach leverages our previous work on the development of SysML-based modeling and analysis techniques for safety-critical systems [4]. Specifically, we tailor the methodology developed in our previous work to the development of safety-critical interfaces, and provide step-by-step and practical guidelines aimed at providing the evidence necessary for arguing that the safety-related requirements of an interface are properly addressed by its design. We describe an application of our proposed guidelines to a representative safety-critical interface in the maritime and energy domain.**

Keywords. Safety-Critical Systems, Software–Hardware Interfaces, Safety Certification, Model Driven Engineering, SysML, Traceability.

I. Introduction

Many safety-critical systems, e.g., those used in the avionics, railways, maritime, and energy sectors, are becoming ever more reliant on software to increase development productivity, enable more complex operations, and provide flexibility in handling evolving needs. While the opportunities and gains offered by the use of software in safety-critical systems are significant, there are also risks associated with the impact and the performance of the software. It is therefore crucial to be able to assess the safety of the software elements of a system throughout the system's entire life-cycle, starting from the inception to operation and all the way to decommissioning. An increasingly common activity performed to this end is *software safety certification*, aimed at providing an assurance that the software-controlled behaviors of a system are deemed safe by a certification body.

Unfortunately, little work has been done to date on accommodating the additional demands that certification imposes on how the design of systems should be expressed. Our experience indicates that certification is often (incorrectly) viewed as an *after-the-fact* activity. This can give rise to various problems during certification, because a large fraction of the safety evidence necessary for certification has to be gathered *during* the design phase and embodied *in* the design specification. Failing to make the design "certification-aware" will inevitably lead to major omissions and effectively make the design "unauditable" for certification purposes.

In the past three years, we have been exploring the use of Model Driven Engineering (MDE) in system design to improve the cost-effectiveness and accuracy of software safety certification. The use of MDE in this context has been motivated by three main principles: (1) Models expressed in standard notations, such as UML [7], SysML [6] and their extensions, avoid the ambiguity and redundancy problems associated with text-based specifications. (2) Models provide an ideal vehicle for preserving traceability and the chain of evidence between hazards, requirements, design elements, implementation, and test cases; (3) Models present opportunities for partial or full automation of many laborious safety analysis tasks (e.g., impact analysis, completeness and consistency checking, and test case generation).

Realizing the full potential of MDE for certification requires methodological guidance that allows safety engineers to understand and apply MDE concepts within their domain of application. In previous work [4], we have developed an MDE methodology based on SysML to enable more efficient auditing of embedded software designs during the certification process. In particular, the framework provides a "certification-aware" methodology for embedded system design along with mechanisms to establish traceability between safety requirements and the design elements. This framework was developed based on observing actual certification projects, consultation with certification experts, and reviewing major safety standards, most notably IEC 61508 [3].

In this paper, we tailor the general framework of [4] for the design of safety-critical interfaces between software

and hardware components (SW/HW interfaces). Note that one often needs to distinguish between an actual SW/HW interface and the software implementation of the SW/HW interface. The latter is what we focus on in this paper, but for brevity, we use the term SW/HW interface (interface, for short where there is no ambiguity) to refer to the software implementation of an actual interface. Interface (implementations) are arguably one of the most complex classes of software components in safety-critical systems: They may need to bridge the timing discrepancies between hardware devices and software control components, and may need to be implemented in different versions so that they can work with different communication protocols, be deployed on different execution platforms, or communicate with different devices. Although interfaces can be small in size, they can have a profound impact on the behavior and stability of embedded systems. As a result, certification bodies regularly ask for accurate design specifications for (SW/HW) interfaces and carefully scrutinize the specifications. Our tailored framework provides a series of concrete and practitioner-oriented design guidelines, aimed at reducing the number and criticality of certification issues related to interfaces.

We illustrate our tailored methodology using a simplified and sanitized version of a real safety-critical interface. For anonymity, we will refer to the interface by the fictitious name of MarineShield (MS). MS is widely used in a certain class of maritime systems for communicating signals from the software control components to a monitoring device. This interface was chosen primarily because its structure and behavior was deemed representative of a large majority of interfaces developed in the maritime and energy sector. As the result of our study, a complete set of design models with traceability to requirements have been developed for MS.

The remainder of this paper is structured as follows: We begin in Section II with a description of the context and motivation for our work. Section III briefly introduces SysML – the language we use for modeling. In Section IV, we describe our tailored methodology for SW/HW interface design and establishing traceability from requirements to design. In Section V, we describe the modeling experience gained over MS and our observations. We conclude the report in Section VI with a summary and a highlight of topics for future work.

II. Context and Motivation

To better understand the difficulties in software safety certification, we observed a number of certification meetings between various leading industry suppliers and certification bodies. Throughout these meetings, we observed that many difficulties could be attributed to the use of text-based documents. Text is prone to ambiguity, incompleteness, and redundancy. This leads to the suppliers and certifiers having to invest a considerable 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 notations would most likely be unfamiliar to the certification bodies. Hence, the diagrams could well become another source of ambiguity, incompleteness, and redundancy.

Further, text is hard 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 represented in standard notations, and not text-based documents, should serve as the main sources of certification information – documents, when needed, should be generated from these models. Our work in this report provides a concrete example of how MDE can be applied in safety certification.

III. Background on SysML

The appeal of SysML in our work comes from the fact that safety-critical software is typically embedded into some greater technical system (one with electronic and mechanical parts). Hence, it is crucial to consider the interactions of software with the non-software elements as well. Since SysML is quickly becoming a de-facto standard for systems engineering [5], it was a natural choice to base our work on. SysML extensively reuses UML 2, while also providing certain extensions to it. Compared to UML, SysML offers the following advantages for embedded systems [6]:

- SysML expresses systems engineering semantics (interpretations of modeling constructs) better than UML, thereby reducing the bias UML has towards software. In particular, UML classes are replaced with a concept called *block* in SysML. Block is a *modular unit of system description*. Blocks are used to describe structural concepts in a system and its environment.
- SysML has built-in *cross-cutting* links for interrelating requirement and design elements. This allows engineers to relate requirements and design elements/models described at different levels of abstraction.

Our methodology in Section IV utilizes four SysML diagrams, namely Block Definition Diagrams (BDDs), Internal Block Diagrams (IBDs), Activity Diagrams (ADs), and Requirement Diagrams (RDs). For a detailed specification of these diagrams and SysML in general, see [1].

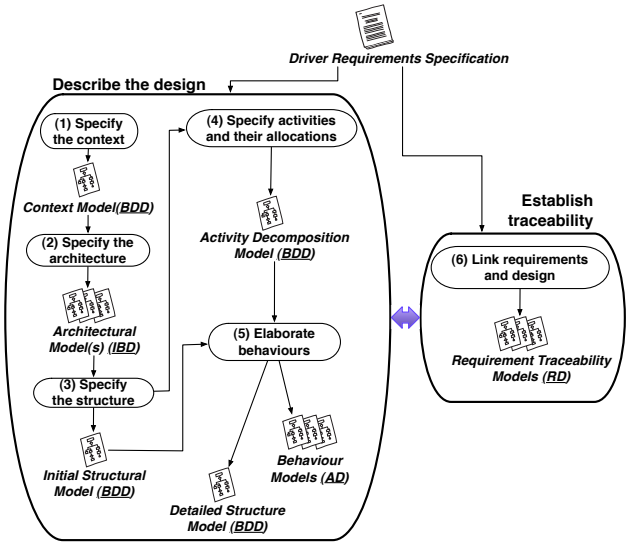

Figure 1. Methodology overview

IV. Methodology for Modeling of Interfaces

In this section, we describe a tailored methodology and concrete guidelines for modeling of SW/HW interfaces, based on a more general methodology developed in our previous work [4]. The main objective pursued here is to facilitate safety certification of interfaces by providing design specifications that are unambiguous and precise at the level required for inspecting how (safety-related) requirements are addressed in the interfaces' designs.

Figure 1 shows an overview of our proposed methodology for model-driven development of SW/HW interfaces. We assume that the requirements for the interface under development have been already specified and provided as input. From a certification standpoint, one may elect to leave out the requirements that have no safety implications. In other words, the requirements specification provided as input in Figure 1 can be restricted to *safety-relevant* requirements, i.e., requirements that in some way contribute to the satisfaction of the overall system safety goals. Our experience indicates however that the significant majority of the interface requirements are safety-relevant. Hence, it seems reasonable to have a complete set of requirements. This ensures that a complete design of the interface will be developed (as opposed to just the safety-relevant design aspects), which is highly beneficial for other development activities such as maintenance, communication and testing.

Our methodology is composed of two parallel but interrelated tasks. The high-level task on the left of Figure 1 ("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 1, 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 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 (Step 6 in Figure 1). If the interface being modeled 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 interface 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 modeled.

In the remainder of this section, we describe each of the 6 steps in the methodology of Figure 1. We illustrate each step using examples from the MS interface[1] and provide guidelines about how to carry out these steps for other interfaces.

Step 1. Specify the Interface's Context

The first step in the modeling of an interface is defining its *context* using a context diagram. This context diagram shows the main system blocks that are related to the interface but are external to it. In SysML, contexts are expressed using Block Definition Diagrams (BDDs). Figure 2 shows the context for MS. As seen from the figure, the `Domain` is defined as being composed of a collection of `ControlComponent`s and a collection of `Interface`s. The `MSInterface` is a specialization of the `Interface` block. It receives input from a particular type of component, named `ComponentX`. Instead of sending the data received from the `ComponentX` directly to the target hardware device, `MSInterface` relays the data to a particular implementation of the FieldBus protocol [2]. We refer to this FieldBus driver implementation as `FBDriverY`. The diagram also captures the fact that both the control components and the interfaces are executed within an overall execution framework, using a `Scheduler`. To implement their function, the interfaces predominantly rely on facilities provided by the Run Time System (RTS) of the operating system. This is modeled using a dependency link from `Interface` to `RTS`.

The diagram in Figure 2 uses three stereotypes: The "block" stereotype denotes SysML blocks. For a given block, the "allocated" stereotype indicates that some requirement, function, activity, etc. has been allocated to the block. We use allocations for assigning activities to blocks (see Step 4). And, the "requirementRelated" stereotype indicates

[1]All element names in the diagrams are sanitized for confidentiality.

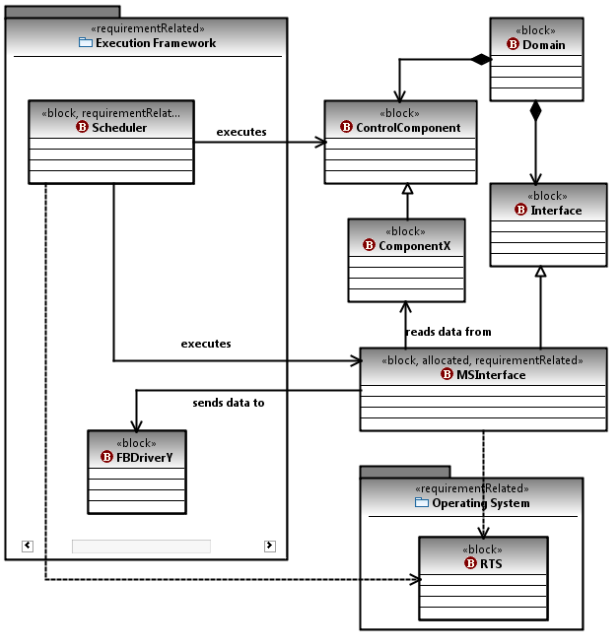

Figure 2. Context for the MS interface described as a SysML BDD

that a given block contributes to the satisfaction of some requirement, i.e., a requirement is traceable to the given block (see Step 6).

Guidelines on Describing Interfaces' Context. The substantial part of the context diagram is shared amongst the interfaces in the whole system and can thus be reused without change. What needs to be revisited for every interface are the following:

1) The control component(s), interface(s), and periphery or third-party software unit(s) that communicate with the interface in question. For `MSInterface`, there is one communicating control component, `ComponentX`, and one periphery software unit, `FBDriverY`.
2) The utility blocks and elements of the execution framework that the interface in question depends on. In the case of `MSInterface`, only some basic facilities in the operating system and the execution framework were used. Other interfaces could have additional dependencies that are not shared by all interfaces. This could require the inclusion of more blocks from the execution framework, the operating system, and periphery or third-party modules, 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 1, where we elaborate the communication links between the interface and other blocks and describe the architectural connections of the interface.

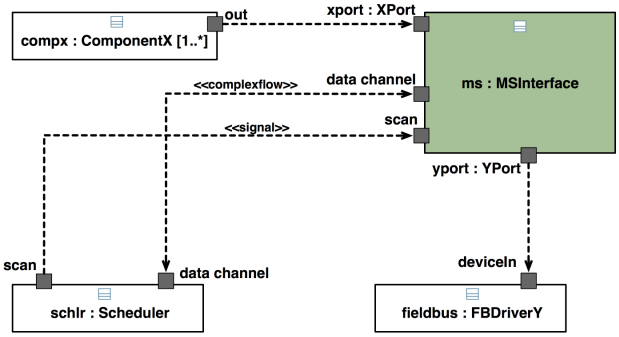

Figure 3. Architectural connections of the interface described in a SysML IBD

Step 2. Specify the Interface's Architecture

In this step, we refine the conceptual relationships that we defined between the interface and other system blocks in the context diagram (Figure 2) into a set of architectural connectors with specific communication ports. Making the links between different system blocks explicit and free from ambiguity is a major concern during safety certification.

We use SysML Internal Block Diagrams (IBDs) for expressing architectural connections. Figure 3 shows the IBD developed for the MS interface. The central component in the diagram is an instance of the `MSInterface` block (named `ms`).

In the IBD, we express several important points about the communication between `ms` and instances of other blocks, particularly: (1) What are the ports for communication? (2) What data and signals are being communicated over the ports? (3) What is the direction of communication? (4) How many instances of each block are participating in the communication? and (5) How many ports (of a certain type) does each block instance have?

The `MSInterface` communicates with three types of blocks: `Scheduler`, `ComponentX`, and `FBDriverY`. 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 3, there is exactly one instance of each block involved except for `ComponentX`, which has at least one but possibly more instances, denoted by the `[1..*]` multiplicity constraint.

Inter-block communications are modeled using ports. The `MSInterface` exposes four types of ports as shown in Figure 3:

- `XPort` for reading data from (one or more) instances of type `ComponentX`. The direction of the arrow from `compx` (instance of `ComponentX`) to `ms` indicates that the communication link is unidirectional. The number of instances of `XPort` that an individual instance of `MSInterface` has depends on how many control components are connected to the interface instance. Hence,

there is a multiplicity constraint of `[1..*]` assigned to the `XPort` type. The modeling 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.

- `YPort` defines a unidirectional port for forwarding the control components' data onto the `FBDriverY`.
- `data channel` is a bidirectional port for communication between `schlr` (instance of `Scheduler`) and `ms`. To indicate that the port combines several information flows, we apply a (user-defined) stereotype "complexflow". The data channel port only handles non-signal data communications, e.g., reading and writing of interface parameters. System control signals (such as the `scan` signal described below) need to be modeled individually and separately. This is because of the important role that these signals play in synchronizing and orchestrating the components of an embedded system. System control signals are frequently referenced in the behavioral design of the components and hence need to be explicitly modeled for certification purposes.
- `scan` is a periodic and global clock signal used for synchronization.

Lastly, we need to note that an IBD captures one specific architectural configuration, not all the possible configurations of an interface. For example, it is possible and also common for an interface to communicate with different components in different deployments. In such cases, each communication architecture is expressed using an individual IBD. For example, an interface 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 an interface would then need to have two IBDs, one for each mode of operation. This is why the methodology in Figure 1 envisages the construction of multiple IBDs. If there is a large or infinite number of configurations, we model only those configurations used in the system being certified.

Guidelines on Elaborating Interfaces' Architecture.

1) Determine if the interface has multiple alternative configurations and communicates with different sets of blocks depending on the configuration. If this is the case, one IBD must be created per configuration. The remaining steps are given for a single configuration (i.e., a single IBD). Generalization to multiple IBDs is straight-forward.
2) Add to the IBD the blocks that directly communicate with the interface. These blocks are readily identifiable from the interface's context diagram. Any block that has an association link to the interface block in the context diagram is included in the IBD. Specify the number of participating instances of each of these blocks using multiplicity constraints.

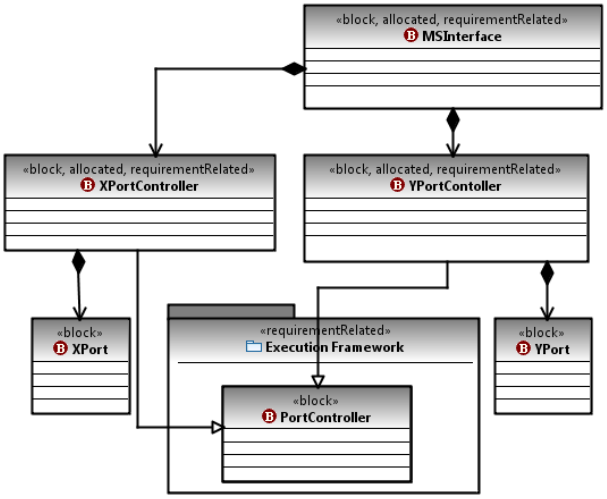

Figure 4. Describing MS's internal structure using a BDD

3) Refine the association links incident to the interface block in the context diagram into information flows in the IBD. Specify the directionality of each information flow. It is possible (and likely) for an association link in the interface'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., `XPort` and `YPort` in MS). These blocks will be included in the interface's structural diagram (see Step 3).

Once an architectural view(s) on the operation of the interface is developed, we can move on to Step 3 where we specify the basic internal structure of the interface.

Step 3. Describe the Interface's Internal Structure

In the previous steps, we defined some of the structural design elements of the interface. In particular, we defined a block representing the interface itself and also blocks for the complex datatypes used by the ports. In this step, we extend the structural design of the interface with controller sub-blocks that encapsulate the behavior of the ports. Most port types, except those denoting primitive signals (e.g., the clock) require an explicit controller.

In Figure 4, we have shown the BDD describing the internal structure of the MS interface. The `XPort`s and `YPort`s are each controlled by their own controller. No controller is created for the (primitive) scan signal.

Guidelines on Elaborating Interfaces' Structure.

1) Define a BDD initially including the interface block from the context diagram and the port data types

specified during the architectural elaboration of the interface (Step 2).

2) For each interface port type from the IBD (developed in Step 2) that has a non-primitive data type:
   a) Define a controller block.
   b) Establish a composition link from the interface block to the controller block, and from the controller block to the port data block.
   c) Add the necessary multiplicity constraints to the composition links, if known. These constraints specify how many instances (minimum, maximum) of a controller block the interface can have, and how many instances of a port a single controller can manage. These multiplicity constraints can be added at later stages of development as well (e.g., when decisions about interface's performance are being made).

Step 4. Specify Interface's Activities and their Allocations

The goal of this step is to specify the activities to be performed by the interface and state how these activities are distributed over the interface's blocks. In Figure 5, we provide a hierarchical decomposition tree for the activities of the MS interface expressed as a BDD. The immediate descendants of the root node (`MS Overall`) represent the interface's main activities. These activities are the creation and deletion of an interface instance (`Create MS Interface`, `Delete MS Interface`), handling of communication with the `Scheduler` (`Process Scheduler Request`), and the interface's core function (`Transfer Data`), which is transferring data from instances of `ComponentX` to the `FieldBus` driver (`FBDriverY`). This last activity itself is decomposed into three finer-grained activities: (1) Reading data from instances of `ComponentX` (`Pull Data from Module`), (2) forwarding the X data to the FieldBus driver (`Push Data onto FieldBus`), and (3) adjusting the data rate for sending information onto the FieldBus (`Monitor Data Rate`).

The leaf activities in the decomposition tree are *allocated* to the structural blocks of the interface. 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 Step 3. More precisely, a leaf-level activity should be small enough to be implementable by one or more operations in 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 5 give rise to two operations in the `MSInterface` block. But not all the operations might be known until the behavioral elaboration in Step 5 is complete.

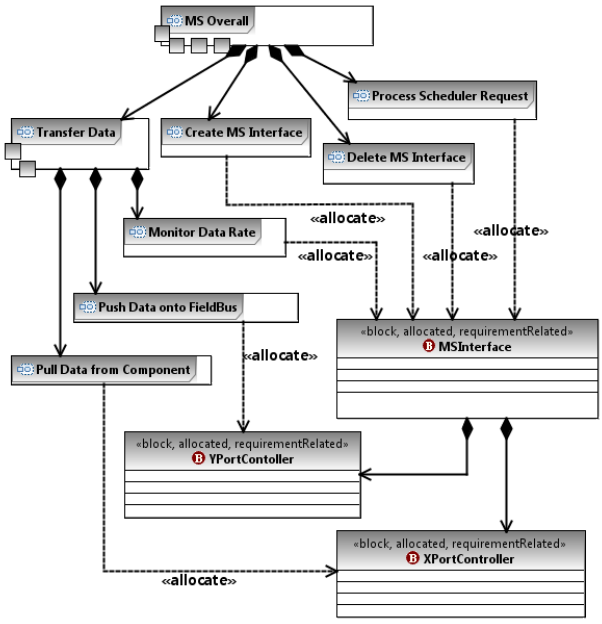

Figure 5. Decomposing MS's activities and allocation to blocks using a BDD

Guidelines on Identifying and Allocating Interfaces' Activities.

1) Specify the top-level activities of the interface. These should at least include: (1) lifecycle activities for creating and deletion of interface instances; (2) high-level activity(ies) to enable the reading and modification of interface parameters; and (3) one or more activities capturing the core functions of the interface.
2) Iteratively decompose the top-level activities. The specifics of the decomposition will vary depending on how the interface is structured and is thus based mainly on human judgment. Generally, decomposition should be necessary only for the core functions.
3) Update the interface's internal structure (initially defined in Step 3) with any newly identified block operations. The structure will be further elaborated in Step 5.

The interactions between activities are not modeled in the activity decomposition tree, nor are concurrency and sequencing of activities. These will be modeled using activity diagrams, as we discuss in Step 5.

Step 5. Elaborate the Interface's Behavior

The goal of this step is to describe the detailed behavior of the interface. During this step, new block operations may be identified as well and added to the structural model previously constructed and refined in Steps 3–4.

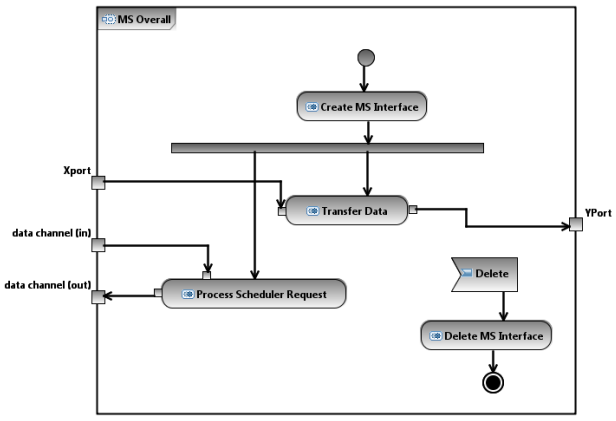

Figure 6. Defining the Interface's overall behavior

SysML offers two different notations for describing the behavioral 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.

Due to space reasons, we only illustrate the application of activity diagrams. The choice of whether to use state machines or activity diagrams (or a combination) depends mainly on what type of behavior is being modeled. For the stateful behaviors of the interfaces, modeling the states and the transitions between them is an important part of the development process. In contrast, stateless behaviors are more naturally captured using activity diagrams.

Figure 6 shows the activity diagram for `MS Overall` – the root activity in the activity decomposition tree of Figure 5. In the figure, we can see the sequencing and interactions between the immediate descendant activities of `MS Overall`. The process begins with the creation of an interface instance. Once an instance has been created, two parallel activities begin: `Process Scheduler Request` and `Transfer Data`. These activities do not terminate until a delete signal is delivered by the `Scheduler`, upon which the delete activity is executed.

The same process illustrated above has to be performed for all composite activities. Once the behavioral elaboration has been performed, the structural model for the interface needs to be refined with the new knowledge from the behavioral design phase. Particularly, this knowledge may lead to the introduction of new blocks (e.g., various types of buffers) to enable communication between the different activities as well as new block operations. In particular, each

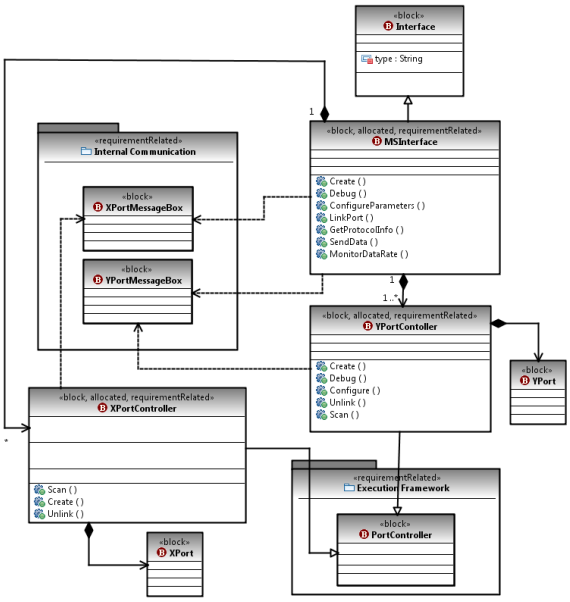

Figure 7. MS's detailed structure expressed as a BDD

activity that is not decomposed any further will be captured as a block operation (e.g., the interface creation activity). Higher-level activities if necessary can be turned into block operations as well (using these more basic operations), but in our example, this was not needed. The BDD capturing the detailed structure of the interface is shown in Figure 7.

Guidelines on Behavioral Elaboration of Interfaces.

1) Define the behavior of each composite activity in terms of its sub-activities.
2) Update the interface's internal structure (initially defined in Step 3 and further refined in Step 4) with any new blocks needed for inter-activity communication and any newly-identified block operations.

To maintain quality and consistency in behavioral 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 remain reachable from the root activity in the activity decomposition tree.
- The input and output of each activity must be specified using parameter nodes. An activity parameter node is an object node at either end of a flow for providing input to an activity or getting output from it [7]. For the root activity in the decomposition tree (Step 4), the parameter nodes correspond to the ports defined in the IBD(s) of Step 2.
- System signals are more suitably modeled as events *within* the activity diagrams rather than activity parameter nodes, e.g., see the `delete` signal in Figure 6.

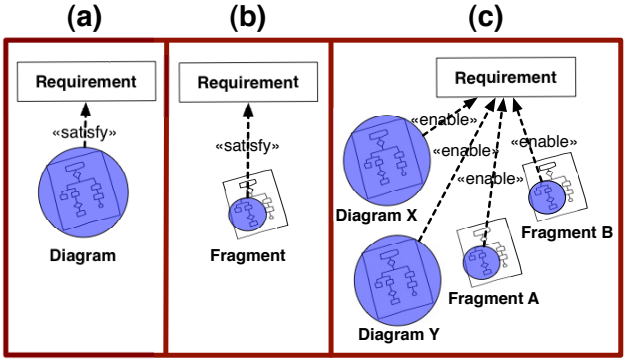

Figure 8. Patterns for traceability links between the requirements and the design

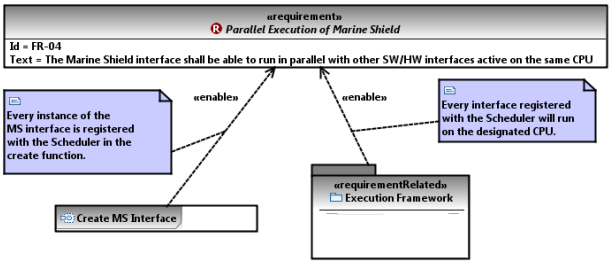

Figure 9. Enabling links from multiple design diagrams to a requirement

Next, we are going to explain how the design models developed in Steps 1–5 above can be linked to the requirements of the interface.

Step 6. Link the Requirements and Design

In this step, we establish traceability links between the interface's requirements and its design using SysML Requirements Diagrams (RD). The traceability links specify which parts of the design contribute to the satisfaction of each requirement. We use three general patterns, shown in Figure 8, for defining the links.

In the first case, Figure 8(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. The second case, Figure 8(b), 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. In the final case, Figure 8(c), we need to deal with the 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.

All three patterns were used in the MS interface, but due to space constraints, we illustrate only case (c), which is the most complex one: one of the interface's requirements states that "The MS interface shall be able to run in parallel with other interfaces running on the same CPU". The execution framework has built-in mechanisms for parallelizing different interfaces on the same CPU and for specifying the multiplicities and the types of interfaces, but for these mechanisms to work, the interfaces have to register with the execution framework. Hence, the satisfaction of the requirement in the diagram will depend both on proper registration by the MS interface during creation as well as the runtime facilities provided by execution framework. The details of the execution framework are outside the scope of our analysis; therefore, the "enable" link in Figure 9 is made from the `Execution Framework` package (which is a fragment of the interface's structural diagram).

V. Experience

We applied our methodology to a real safety-critical SW/HW interface and built a complete SysML design with traceability to the interface's requirements. The resulting design and traceability links were iteratively validated and refined in collaboration with a domain expert (last author).

The SysML design in this study includes all the SysML diagrams envisaged in our methodology. Specifically, the design consists of 23 diagrams, 194 elements having 186 relations and 57 attributes. The interface under study included 30 requirements all of which were safety-relevant. All these requirements were related to the SysML design using appropriate traceability links. Through the application of our methodology to the MS interface we aimed to investigate two questions: **Q1:** Is our methodology applicable in a realistic setting? **Q2:** Does the methodology address the needs of design audits during certification?

Q1. Throughout the design, we created the following SysML diagrams: one context diagram (BDD), one architecture diagram (IBD), one detailed structure diagram (BDD), one activity decomposition diagram (BDD), one overall behavior model (AD), and 19 detailed behavior models (ADs). To trace the interface's requirements to the design, a total of 65 traceability links were created manually for the 30 requirements (RDs). The entire study (design and establishing traceability) was done over three weeks, involving approximately 40 man-hours of effort. This was considered worthwhile because the developed models provide a precise and convenient way for exchanging information within a team, with suppliers, and last but not least with the certifiers. Further, as the certification process and all its associated activities are costly and may take a long time even for small SW/HW interface implementations, three weeks is very little in comparison. A yet another perceived benefit of the models

Table I
SOME RECURRING ISSUES RAISED DURING CERTIFICATION.

Issue	Certifier's Demand
Relationships between the software blocks and whether they are at the same or different levels (subsystem, component) are not clear.	Provide a diagram to show the decomposition of the software system into its constituent blocks.
Interactions between components and subsystems have not been made explicit.	A detailed description of the architecture must be developed. The interfaces between software components, input and output bits for terminals, and logical information flows between terminals must be specified.
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.
Application workflows are difficult to understand from the (textual) descriptions provided.	A sequence diagram or an activity diagram needs to be provided for easier understanding of the workflows.
It is hard to identify which blocks are involved in meeting each requirement.	Traceability between requirements and design must be clearly stated.
Different modes of the system and admissible transitions between different modes are not documented.	Different states in the system, and the conditions and events for moving from one mode to another must be captured using a model.

is that they will simplify impact analysis on modifications made at a later stage.

Q2. Table I represents a list of recurring issues raised during the certification meetings we attended (see Section II), and the certifier's demands. The application of our methodology would have avoided the majority of the observed issues noted in Table I by prescribing what kind of diagrams must be created, guiding engineers to iteratively define and expand these diagrams, providing heuristics about the level of detail that must be included, and giving explicit guidelines on creating traceability links.

VI. CONCLUSION

Building on our previous work on SysML-based design of safety-critical embedded systems [4], we developed in this paper a set of methodological guidelines specifically aimed at SysML-based modeling of safety-critical SW/HW interfaces. Our primary focus was improving the safety certification process, by avoiding recurrent design specification issues identified in the design inspections and audits conducted by the certification bodies. As a case study and a way to validate our methodology, we built a detailed SysML design for a real safety-critical interface for maritime and energy systems.

Within the work conducted so far, there are several topics that need further investigation. Key topics for future research include:

Writing better requirements: Design quality is necessarily influenced by the quality of the requirements. We plan to develop guidelines on writing and classification of interface requirements. This will result in both a more accurate design, and more precise requirements-to-design links.

Non-functional requirements: Our work is currently limited to functional requirements. We are now working on generalizing our approach to non-functional requirements (e.g., performance and availability).

Further case studies: While we have attempted to ensure as much generalizability as possible through working on representative examples, we are aware of the diversity of the SW/HW interfaces used in different systems and different domains. Our focus has been mainly on systems in the maritime and energy sector. The extent to which our work applies to other domains, and whether the resulting models provide enough detail for certification audits in these other domains requires further investigation which we plan to do in the future. Also, although our methodology is aligned with real problems observed during certification, we have not yet conducted a thorough evaluation of the usefulness of the resulting models during certification. This too will be tackled as part of our future work.

REFERENCES

[1] S. Friedenthal, A. Moore, and R. Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann, 2008.

[2] Iec 61158: Industrial communication networks - fieldbus specifications. International Electrotechnical Commission, 2010.

[3] IEC 61508: Functional safety of electrical / electronic / programmable electronic safety-related systems. International Electrotechnical Commission, 2005.

[4] S. Nejati, M. Sabetzadeh, D. Falessi, L. Briand, and T. Coq. A SysML-based approach to traceability management and design slicing in support of safety certification: Framework, tool support, and case studies. *Information and Software Technology (forthcoming)*, 2011. http://modelme.simula.no/assets/IST11.pdf.

[5] W. Schafer and H. Wehrheim. The challenges of building advanced mechatronic systems. In *FOSE '07*, pages 72–84, 2007.

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

[7] UML 2.0 Superstructure Specification, 2005.