A SysML-Based Approach to Traceability Management and Design Slicing in Support of Safety Certification: Framework, Tool Support, and Case Studies

Shiva Nejati[1] Mehrdad Sabetzadeh[1] Davide Falessi[1] Lionel Briand[1] Thierry Coq[2]

[1]*Simula Research Laboratory, Norway* {shiva,mehrdad,falessi,briand}@simula.no

[2]*Det Norske Veritas, France* thierry.coq@dnv.com

Abstract

Context: Traceability is one of the basic tenets of all safety standards and a key prerequisite for software safety certification. In the current state of practice, there is often a significant traceability gap between safety requirements and software design. Poor traceability, in addition to being a non-compliance issue on its own, makes it difficult to determine whether the design fulfills the safety requirements, mainly because the design aspects related to safety cannot be clearly identified.
Objective: The goal of this article is to develop a framework for specifying and automatically extracting design aspects relevant to safety requirements. This goal is realized through the combination of two components: (1) A methodology for establishing traceability between safety requirements and design, and (2) an algorithm that can extract for any given safety requirement a minimized fragment (slice) of the design that is sound, and yet easy to understand and inspect.
Method: We ground our framework on System Modeling Language (SysML). The framework includes a traceability information model, a methodology to establish traceability, and mechanisms for model slicing based on the recorded traceability information. The framework is implemented in a tool, named SafeSlice.
Results: We prove that our slicing algorithm is sound for temporal safety properties, and argue about the completeness of slices based on our practical experience. We report on the lessons learned from applying our approach to two case studies, one benchmark and one industrial case. Both studies indicate that our approach substantially reduces the amount of information that needs to be inspected for ensuring that a given (behavioral) safety requirement is met by the design.

Keywords: Safety Certification, SysML, Traceability, Model Slicing

1. Introduction

Safety-critical software is typically subject to a strict safety certification process. The goal of the process is for a licensing or regulatory body to review the safety evidence and arguments provided by the supplier and ensure that the development and usage of the software are in compliance with the applicable safety standards, e.g., IEC 61508 [1] (and its sector-specific specializations) for various kinds of programmable devices, DO-178B [2] for airborne systems, and the upcoming ISO 26262 [3] for the automotive industry.

Traceability is one of the core principles mandated by all these safety standards, with an overarching effect on all aspects of development. Typically, the development of a safety-critical system begins with hazard and risk analysis. The results of this analysis are used to define the overall (system-level) safety requirements. The safety requirements for the "software" elements of the system are derived from the overall safety requirements and realized within the software design and implementation. In such a development context, it is essential to preserve traceability from hazards and risks to the overall safety requirements on to the software safety requirements on to the software design on to the software implementation. Further, the development artifacts built throughout the process must be traceable to the various verification and validation activities (e.g., static analysis, testing, formal proofs) that they have been subject to. This web of traceability information is not only crucial for the maintenance and evolution tasks to be performed by the supplier, but is also a key prerequisite for any systematic inspection of software safety by the certifiers.

This present article was prompted by the difficulties in the software safety certification process that arise from poor traceability. These difficulties were observed during an investigation of the software safety certification inspections in the maritime and energy industry, but the problems should also be representative of those faced in other software-intensive safety-critical domains, such as the automotive industry, where software safety certification is an emerging topic.

An important source of traceability problems in software safety certification is the chain from the overall safety requirements to software safety requirements to software design. In the current state of practice, this chain often lacks sufficient detail to support the inspections that certifiers conduct to ensure that the (software-related) safety objectives of a system are properly addressed by the software design. Poor traceability, in addition to being a non-compliance issue on its own, makes it hard to check whether the design satisfies the safety requirements. This is largely due to the safety-related design aspects not being clearly identifiable.

1.1. Contributions

Broadly, this article is concerned with the problem of traceability from requirements to design. This problem is driven by three main considerations: (1) the way the requirements are expressed, (2) the way the design is expressed, and (3) the goal to achieve from traceability. In our work, the requirements are expressed as (unrestricted) natural language statements; the design is expressed using the Systems Modeling Language (SysML) [4]; and the goal is to facilitate design safety inspections by enabling engineers to focus on the aspects of the design that are relevant to safety. This goal is achieved by automatically extracting, for each safety requirement, a minimized fragment (slice) of the design that is sound, and yet easy to understand and inspect. The combination of natural language requirements, SysML, and slicing is novel and useful in practice, noting that SysML is an INCOSE [5] standard and represents a significant and increasing segment of the safety-critical software industry. Our work in this article concentrates specifically on *behavioral* safety requirements, i.e., safety requirements that constrain some system behavior. We do not consider here other categories of requirements, such as performance, availability, and security, which can have safety implications. Specifically, we make the following contributions in this article:

- We characterize, based on our observation of actual certification meetings, consultation with certification experts, and reviewing major safety standards (most notably IEC 61508 [1]), the traceability information that is necessary for arguing that the (behavioral) safety requirements are accounted for in the software design. This is achieved by developing an *information model* for traceability.
- We develop a SysML-based methodology to guide designers in establishing the traceability links prescribed by our information model.
- We devise an algorithm that, using the established traceability links, automatically extracts slices relevant to a (behavioral) safety requirement, thus reducing the amount of information to be inspected.
- We develop a tool to support our methodology and slicing algorithm. The tool, named SafeSlice (`http://modelme.simula.no/pub/pub.html#ToolSlice`) [6], helps designers follow the methodology, automatically checks the consistency of the established traceability links, and automatically generates SysML design slices based on the links. Additionally, the tool provides facilities for model navigation, project status monitoring, and report generation.

- We provide both formal and empirical validation of our work. In particular, we provide a formal proof that our slicing algorithm is sound for behavioral safety properties, and argue about the completeness of the generated slices based on our experience. We report on the lessons learned from applying our approach to two case studies, one a benchmark case study and the other an industrial case study concerning a safety Input/Output (IO) software module used on ships and offshore facilities [7]. Both case studies indicate that our approach offers benefits by substantially reducing the amount of information that needs to be inspected in order to ensure that a given safety requirement is met by the design.

1.2. Structure

The remainder of the article is structured as follows: In Section 2, we provide a short overview of SysML. We describe our SysML-based methodology in Section 3. The information model upon which the traceability links in the methodology are based is given in Section 4. In Section 5, we discuss how the traceability links can be utilized for automation, specifically, for automatic extraction of the design information relevant to a particular safety requirement. Section 6 presents tool support. Section 7 provides an evaluation of our approach. Section 8 reviews the related work. Finally, Section 9 summarizes the article and outlines directions for future work.

2. Background on SysML

In this section, we provide a brief introduction to SysML and highlight its main advantages. The appeal of SysML in our work comes from the fact that safety-critical software is typically embedded into some greater technical system (e.g., 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 rapidly becoming a de-facto standard for systems engineering [8], it was a natural choice to base our work on.

SysML extensively reuses UML 2, while also providing certain extensions to it. 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 continuous constraints for property values of hardware components. Activity Diagrams, Internal Block Diagrams, and Block

Definition Diagrams are modifications of existing Activity Diagrams, Collaboration Diagrams, and Class Diagrams in UML 2, respectively. Sequence Diagrams, State Machine Diagrams, Use Case Diagrams, and Package Diagrams in SysML are identical to their UML 2 counterparts.

Compared to UML, SysML offers the following advantages for specifying control systems [9]:

- SysML expresses systems engineering semantics (interpretations of modeling constructs) better than UML, thus 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 3 utilizes all SysML diagrams. For a complete specification of SysML, see [10].

3. Design Methodology

SysML is only a modeling notation and does not provide any specific modeling methodology. To use SysML effectively, one needs a methodology tailored to the problem domain and the objectives to be achieved from modeling. In this section, we describe a SysML-based methodology for modeling safety-critical control systems. Our methodology adapts the best practices in the existing model-based systems engineering methods [11] to the needs of software safety certification. The main objective pursued here is providing precise and unambiguous specifications of the system requirements and design, and establishing traceability between the two. The traceability provides the basis for clearly identifying the design aspects that are related to the safety requirements. This in turn allows safety engineers to more effectively check whether the safety requirements are addressed by the design.

We use a small fragment of a Production Cell System (PCS) [12] as a working example. Briefly, the aim of PCS is the transformation of metal blanks into forged plates (by means of a press) and their transportation from a feed belt into a container. We focus on interactions between two devices of the cell: the feed belt and the (rotary) table. The cell operator puts the blanks one by one on the feed

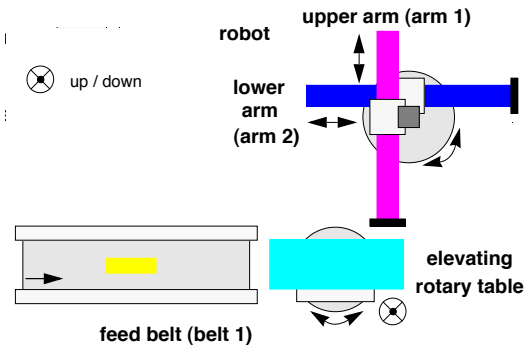

Figure 1: A fragment of the Production Cell System (PCS) [12]

belt and the belt conveys them to the table. The table then rotates and lifts to put the blanks in the position where a robot arm can take them. A picture of the feed belt, table, and the robot arms is shown in Figure 1. Several safety requirements are stated in the specification of PCS to ensure its safe operation including the one below:

Avoidance of falling metal blanks: *The metal blanks must not be dropped outside safe areas of PCS. The safe areas include the surface of PCS devices and places that are reachable by the robot arms.*

Figure 2 shows an overview of our proposed methodology. The methodology includes one or more iterations of the following two main phases: In the first phase, the system requirements are identified. The second phase is composed of three parallel but inter-related tasks: Describe the system structure and Describe the system behavior that are concerned with the construction of structural and behavioral design models respectively, and Establish Traceability which is concerned with the creation of traceability links between the requirements and design.

The input to our methodology is (1) a set of standards for the domain of the system under analysis, (2) stakeholders' requirements, and (3) a model capturing domain concepts and their relationships. In Figure 2, the steps within each phase are depicted as being conducted sequentially. However, in reality, the discoveries made at later stages of the development may affect the decisions made in earlier stages. Thus, the diagrams developed in the process will co-evolve and none will be considered final until the design is complete.

Our methodology borrows and adapts from other Model Driven Engineering (MDE)-based methodologies in particular, from [13] and [14], and from the MDE

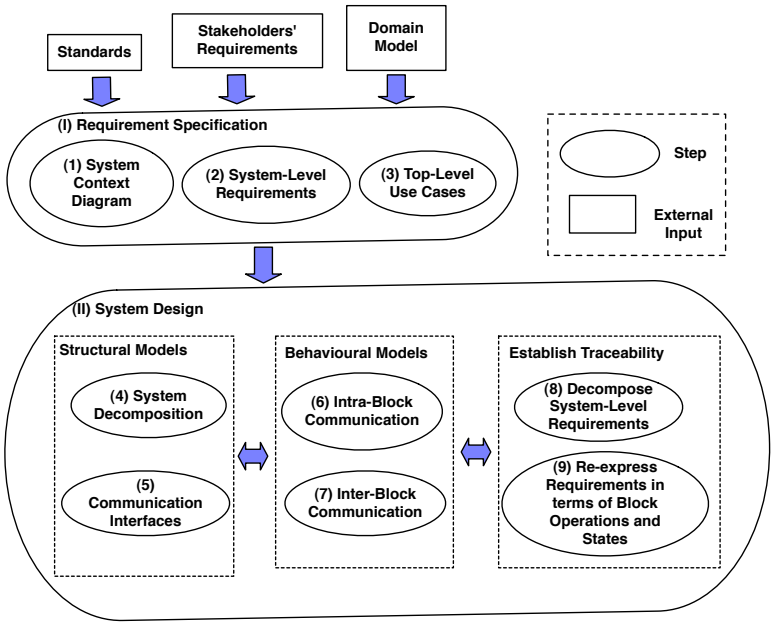

Figure 2: Methodology for model-driven development of control systems.

methodologies for systems engineering described in [11, 10]. The two main characteristics that distinguish our methodology from the previous work are:

1. The decomposition of system-level requirements is interleaved with the design steps rather than preceding them. The reason is that requirements decomposition implicitly contains a decomposition of the system into its subsystems and components. Hence, unless some thought is given to the system design first, decomposing the requirements may cause a premature commitment to decisions about the system structure. To avoid this problem, we suggest that an initial decomposition of the system precedes the derivation of the lower-level requirements.
2. The guidelines for capturing requirement and design details and creating traceability links from requirements to design are specific for safety certification inspections.

Below, we briefly describe the steps comprising our methodology, emphasizing the guidelines related to safety certification inspections. A complete description of our methodology is available in [15, 16].

Phase I - Requirements Specification. As shown in Figure 2, this phase has

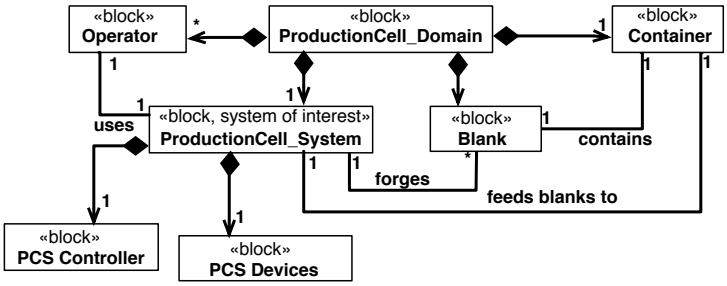

Figure 3: Context diagram for PCS.

three steps:

Step 1: System context diagram. The purpose of the system context diagram is to specify the boundary between the system and its context. The system context typically includes users, and hardware/software sub-systems that directly interact with the system [13]. We use SysML blocks to represent the context. As an example, the system context diagram for PCS is shown in Figure 3.

Guidelines for constructing a system context diagram: Create a Block Definition Diagram (BDD) as follows:

- Define a top-level block indicating the domain.
- Decompose the domain block into a block, denoting the system of interest, and the blocks that directly interact with the system of interest, e.g., users and hardware/software subsystems. In our methodology, we use the *system of interest* stereotype for the block representing the system that we are studying.
- Describe the relationships between the system of interest block and the other blocks.
- Decompose the system of interest block into a software controller block and a block denoting hardware/mechanical devices.

Many safety requirements arise from assumptions about the system context, also known as *environmental assumptions*. Incorrect environmental assumptions and incorrect transition from these assumptions to system requirements may cause catastrophic system failures [17]. Therefore, it is important to specify these assumptions explicitly and in an analyzable form. We describe the environmental

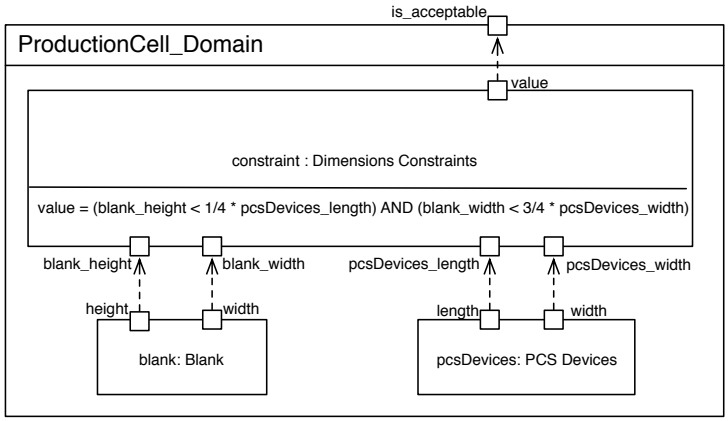

Figure 4: A SysML parametric diagram expressing an assumption.

assumptions using SysML parametric diagrams for continuous (numeric) properties of a hardware entity, and OCL constraints for discrete (logical) properties of a software entity. In addition, we create traceability links from assumptions to system-level requirements.

In Figure 4, we have shown an example parametric diagram. The diagram describes a domain assumption about the physical dimensions of the blanks that are fed to PCS. The assumption states that the height of a blank is no larger than 1/4 of the length of the PCS devices, and that the width of a blank is not larger than 3/4 of the width of the PCS devices. These constraints ensure that the blanks can be carried and transferred safely by the PCS devices from the feed belt to the press and from there to the container.

Guidelines for describing environmental assumptions (Parametric Diagrams):

- Describe a constraint relating the physical properties of the hardware-related blocks in the context diagram.
- Specify the input and output variables of the constraint. Map the input variables of the constraint to the variables of the blocks in the context diagram, and specify the output variables as the output of the parametric diagram.

Guidelines for describing OCL constraints are available in [14].

Step 2: System-level requirement diagram. System-level requirements address the entire system, and can relate to both hardware and software. These requirements typically describe the needs of the users that interact with the system and are defined at the level of the blocks in the system context diagram. System-level requirements may come from a variety of sources including, standards, cus-

tomers, domain experts, environmental assumptions. Our focus here is to capture (1) *safety requirements*, i.e., quality requirements ruling out software effects that might result in accidents, degradations, or losses in the environment [18] (e.g., the PCS requirement described earlier in this section), and (2) *safety-relevant requirements*, the requirements that in some way contribute to the satisfaction of the system safety requirements. Examples of system-level (safety) requirements from the PCS include:

Restrict machine mobility. Each PCS device should be stopped before the end of its possible movement, otherwise it would destroy itself.

Avoid machine collisions. There should not be any collision between PCS devices.

Avoid falling metal blanks. The blanks must not fall on the ground.

Avoid piling or overlapping blanks. Blanks should not be piled on each other, overlapping, or placed too close for being distinguished by PCS photoelectric cells.

Step 3: Use case diagram capturing system's top-level functions. Use cases represent the functionality of the software part of a system from an external point of view. In our methodology, like many existing model-based methodologies [14, 10], use case diagrams are used to represent an overview of the system functions that are later refined into detailed design views. They can be directly traced to the system-level requirements that they are expected to address. In our work, use case diagrams can be created following the guidelines presented in [14, 10]. The use case diagram for PCS is shown in Figure 5.

Phase II - System Design. Software design involves the creation of two main complementary views: structural and behavioral. Structural views describe organization of a system in terms of its constituent blocks and their interaction points, and behavioral views describe how the blocks work together and communicate with one another to deliver functionality. Below, we first briefly describe the diagrams capturing these two views, and then discuss the Establish Traceability task.

Steps 4,5: Structural Diagrams. We describe the structure of a system using SysML Block Definition Diagrams (BDDs) and Internal Block Diagrams (IBDs). The BDDs are used to represent the system hierarchical decomposition (Step 4 in Figure 2), and the IBDs are used to represent the communication interfaces

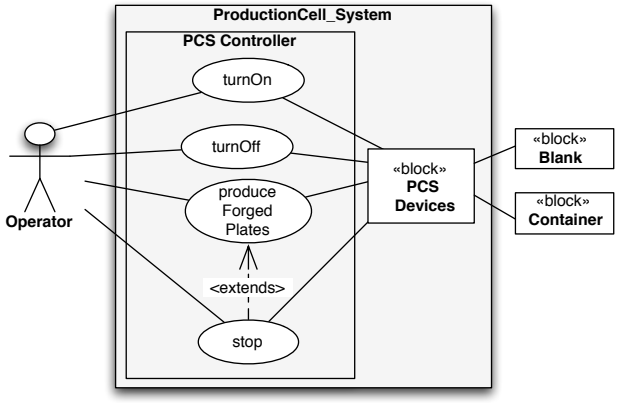

Figure 5: Use case diagram for PCS.

between system blocks (Step 5 in Figure 2) [10]. Note that we also used BDDs for the purpose of creating system context diagrams in Step 1.

System Decomposition BDDs are used for decomposing a system into its constituent blocks and specifying conceptual relations between these blocks. We use *association* relations to represent communication between software and hardware blocks, and *association*, *dependency* and *generalization* relations to model conceptual relations between software components. Figure 6(a) shows a fragment of the BDD for the PCS controller introduced at the beginning of this section. This diagram shows the decomposition of the PCS controller into software blocks related to the feed belt and table devices.

Guidelines for creating system decomposition diagrams (BDD):

- Define a BDD that initially includes, from the system context diagram developed in Step 1, the system of interest block and its constituent blocks: the controller block, and the hardware devices block.
- Decompose the controller block (resp. the hardware devices block) into specific software (resp. hardware) blocks. The decomposition stops when all the blocks required for establishing traceability links in Steps 8 and 9 are generated.
- Specify interactions between software and hardware blocks using association relations, and specify interactions among software blocks using association, generalization and dependency relations.
- Add the necessary multiplicity constraints to the relations.

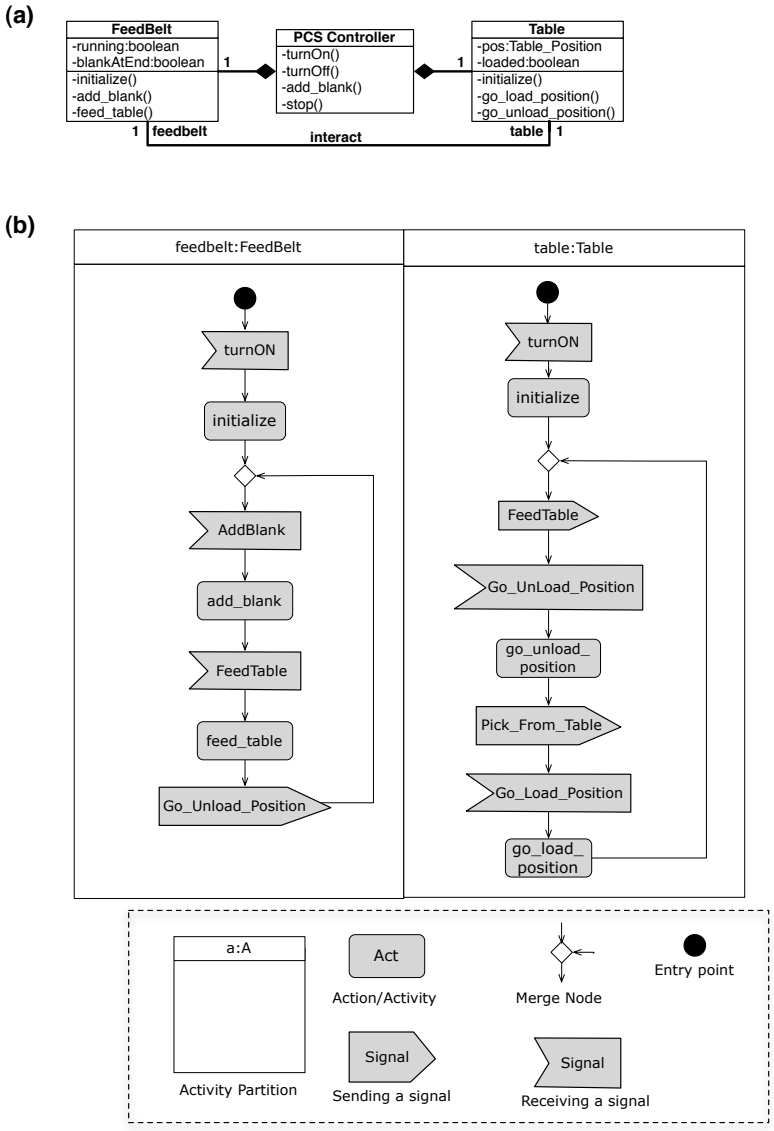

Figure 6: A fragment of design diagrams for PCS: (a) Block Definition Diagram, and (b) An activity diagram consisting of two activity partitions where each corresponds to a block in (a).

Communication Interfaces IBDs are used for specifying communication between the blocks identified in BDDs. More specifically, we refine the conceptual relationships that we defined between the software/hardware blocks in a BDD into a set of architectural connectors with specific communication interfaces and ports. Making the interfaces between different system blocks explicit is a ma-

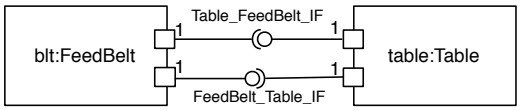

Figure 7: An fragment of the internal block diagram for PCS representing interaction points between software blocks and their interfaces.

jor concern in safety-critical systems to ensure that components can be integrated properly. We capture two kinds of communications using IBDs: (1) Communications between the hardware and software blocks, and (2) Communications between software blocks (Figure 7 represents an example of the latter kind of communications).

Guidelines for capturing communications between hardware and software blocks (IBD):

- Create an IBD by including the software and hardware parts from BDD that directly communicate with one another.
- Refine the association relations between software and hardware blocks in the BDD into information flows in the IBD. For each flow, specify the communication ports on both ends. Define the multiplicity for each port.

Guidelines for capturing communications between software blocks (IBD):

- Create an IBD by including the software parts from BDD that participate in a use case (Step 3).
- Refine the relations between software blocks in the BDD using interaction points specified through standard ports. At each interaction point, a part either *provides* or *requires* an interface. Define the multiplicity for each port.

Steps 6,7: Behavioral Diagrams. Like many existing model-based approaches, we use sequence/activity diagrams to represent *inter*-block scenarios, and state machine diagrams to represent *intra*-block state-based specifications [19]. For each software block with control behavior, we create one state machine diagram. Each such block can be further related to a timeline or a partition in some sequence or activity diagram, respectively. We make these relations explicit by creating SysML allocation links from each block to its related state machine, and to its related sequence diagram timelines and activity partitions. In

the rest of the article, we focus on activity diagrams for representing block behaviors. Guidelines for creating sequence diagrams and state machine diagrams, and examples for these diagrams are available at [15, 16].

Figure 6(b) includes a small legend summarizing the notational elements used in our activity diagrams. As shown in the figure, an activity diagram consists of several activity partitions that capture the parallel behaviors of their corresponding blocks. Each activity partition contains a sequence of nodes for actions/activities, and nodes to represent sending and receiving of signals. In our methodology, we represent the communication between different activity partitions via message passing where messages capture signals. This mode of communication is common in embedded software systems. Note that our treatment for activity diagrams can be easily generalized to other behavioral diagrams (see [15]).

Using behavioral diagrams, we can infer *temporal* dependencies between system operations, i.e., we can identify the relative ordering of the occurrence of system operations. Specifically, from activity diagrams, we can infer the following dependencies: (1) Triggering of an activity/action by a signal. For example, the `FeedTable` signal triggers the `feed_table` activity in Figure 6(b). (2) Sending of a signal upon completion of an activity/action. For example, the `go_load_position` activity triggers sending of the `FeedTable` signal. (3) Communication of a signal from one activity partition to another activity partition. For example, the `FeedTable` signal sent from the activity partition related to `Table` is received by the activity partition related to `FeedBelt`.

Note that call relations between block operations can be reflected to relations between activities/actions related to those block operations. For example from the activity diagram in Figure 6(b), we can infer that the `go_load_position`() operation of `Table` triggers the `feed_table`() operation of `FeedBelt` because in their related activity partitions, upon completion of the `go_load_position` activity, `Table` sends the `FeedTable` signal to `FeedBelt` which, in turn, triggers the `feed_table` activity.

Guidelines for creating behavioral diagrams (Activity Diagrams):

- Identify from the IBD diagram, that captures the communications between software blocks (Step 5), the blocks that interact with one another. Create activity partitions for each of these blocks.

- Identify the sequence of actions that each block performs and complete activity partitions for each block.

- Specify the communications between blocks by signals that are sent and received across activity partitions.

Consistency: To keep the activity diagram consistent with the structural diagrams, the following rules must hold:

- Each block with control behaivour must be related to an activity partition.
- The actions in each activity partition must appear as operations in the block that the partition represents.
- The received (resp. sent) events in each activity partition should be part of the *provided* (resp. *required*) interface of the block represented by the partition.

For example, Figure 6(b) shows the activity partitions related to `FeedBelt` and `Table` blocks in Figure 6(a). The actions `initialize`, `add_blank`, and `feed_table` in the activity partition related to `FeedBelt` appear as `FeedBelt` operations. Similarly, the actions `initialize`, `go_load_position`, and `go_unload_position` in the activity partition of `Table` appear as `Table` operations in Figure 6(a). A complete set of consistency rules between behavioral and structural diagrams used in our methodology is available at [15, 16].

Steps 8,9: Establish traceability. Through the activities under this task, we establish traceability links from the system-level requirements down to the design diagrams adapting and using the SysML traceability links. The traceability links specify which parts of the design contribute to the satisfaction of each requirement. This part of the methodology is the main extension compared to the existing methodologies [13, 14, 11, 10]. We expect the engineers to undertake the activities under this task only for selected safety and safety-relevant requirements that are subject to stringent inspections during certification. Our approach to establishing traceability links has the following two steps:

From system-level requirements to block-level requirements. In this step, we decompose system-level requirements into lower-level requirements that can be traced to a single or a small set of blocks contributing to the satisfaction of that requirement. Decomposition of system-level requirements structurally mimics the decomposition of the system into its constituent blocks discussed in Step 4.

We create explicit links between system-level and block-level requirements using SysML *decompose* links, and between block-level requirements and their related blocks using SysML *trace* links. For example, Figure 8 shows how a

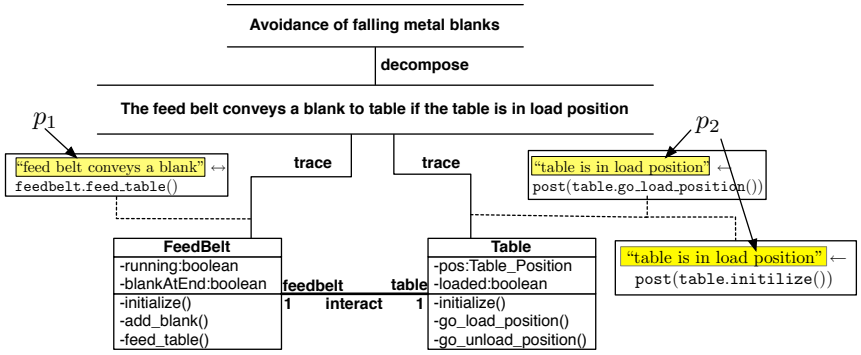

Figure 8: Traceability from a system-level requirement to a block-level requirement and from the block-level requirement to the relevant blocks.

system-level requirement of the PCS example is decomposed into a block-level requirement which is traced to `FeedBelt` and `Table` blocks from Figure 6(a).

One way to assist engineers in decomposing system-level requirements is to first trace down the requirement to use cases, and then further down to sequence diagrams related to those use cases. Some of these blocks whose instances appear in the sequence diagrams are responsible for fulfilling system-level requirements. The next step is then to derive block-level requirements for these blocks by carefully analyzing scenarios, their messages, and triggered operations.

Re-express requirements in terms of block operations and states. The *trace* links specified in Step 8 are rather too coarse-grained because requirements often do not concern entire blocks. Rather, they refer to particular operations or states of the blocks. We make the traceability links to blocks more specific by augmenting them with mappings from the requirement phrases to block operations or to block states. Syntactically, the mappings are generated by the following simple grammar:

$$
\begin{array}{lll}
\textit{mapping} & ::= & \texttt{phrase\_from\_requirements}\ \textit{rel}\ \texttt{block\_op} \mid \\
 & & \texttt{phrase\_from\_requirements}\ \textit{rel}\ \texttt{block\_st} \\
\textit{rel} & ::= & \leftrightarrow \mid \rightarrow \mid \leftarrow
\end{array}
$$

where, *mapping* and *rel* are non-terminals, and the rest, which are terminals, are explained below. We use the requirements and blocks in Figure 8 for exemplification.

- `phrase_from_requirements` is a requirement phrase, describing one of the following situations:

1. an action being performed by a system block, e.g., "feedBelt conveys a blank"; or
2. a state or period during which a block is stable, i.e., block attributes do not change their values, e.g., "table is in load position".

- `block_op` denotes a block operation and is formalized as `block.blockop()`, e.g., `feedbelt.feed_table()`.
- `block_st` denotes a boolean expression describing a block state and can be formalized in the following ways:

  1. as a state invariant of a block: $(\texttt{block.attr}_1 = \texttt{v}_1 \wedge \ldots \wedge \texttt{block.attr}_\texttt{n} = \texttt{v}_\texttt{n})$ e.g., `table.pos` $=$ `loadposition`; or
  2. as a pre condition of a block operation: `pre(block.op())`, e.g., `pre(feedbelt.feed_table())`; or
  3. as a post condition of a block operation: `post(block.op())`, e.g., `post(feedbelt.feed_table())`.

  Note that `pre(feedbelt.feed_table())` and `post(feedbelt.feed_table())` describe the state of the FeedBelt block before and after execution of `feed_table()`, respectively.
- $\leftrightarrow$, $\rightarrow$, $\leftarrow$ are implication relations describing how a `phrase_from_requirements` is related to a block operation or a block state. Specifically, we use $\leftrightarrow$ when the situation described by `phrase_from_requirements` is fully captured by `block_op` or `block_st` on the right-hand side, and $\rightarrow$, $\leftarrow$ when `phrase_from_requirements` respectively describes a less general or more general situation than the right-hand side.

The guidelines for creating the above mappings are as follows: (1) Decompose the requirement into phrases referring to actions or states of a system. (2) Determine block operations and block states related to the phrases. (3) Use logical implication relations introduced above to relate each phrase to a block operation or a block state.

Suppose we want to augment the trace links in Figure 8 with mappings. The requirement in Figure 8 has two phrases: $p_1 =$ "feedBelt conveys a blank" referring to a block action, and $p_2 =$"table is in load position" referring to a block state. The phrase p_1 matches the `feed_table()` operation of FeedBelt, as this operation is responsible for passing the blank to the table, and the phrase p_2 refers to a state

where the Table block is in loadposition. This state can be formalized in several ways: (1) via the state invariant table.pos = loadposition, (2) via the post condition post(table.go_load_position()) as the operation go_load_position() causes the table to move to its load position, (3) via the pre condition pre(table.go_unload_position()) as the operation go_unload_position() assumes that the table is already in its load position, and (4) via the post condition post(table.initialize()) as the operation initialize() causes the table to move to its initialized position which is the load position. We then use logical implication relations to establish the mappings between p_1, p_2 and block operations and states:

(1) $p_1 \leftrightarrow$ feedbelt.feed_table()
(2) $p_2 \leftrightarrow$ table.pos = loadposition
(3) $p_2 \leftarrow$ post(table.go_load_position())
(4) $p_2 \leftarrow$ pre(table.go_unload_position())
(5) $p_2 \leftarrow$ post(table.initialize())

In (1) and (2), the phrases are equivalent to the left-hand side expressions, but not in (3), (4), and (5). Informally, (3), (4), and (5) hold because table being in load position is one of the conjuncts in post(table.go_load_position()), pre(table.go_unload_position()), and post(table.initialize()), respectively.

In Figure 8, we use mappings (1), (3), and (5) to augment the links from the requirement to FeedBelt and Table, respectively. We could further add mappings (2) and (4) to our example. The exact choice of the mappings to use depends on the logical argument that the designer wants to provide to demonstrate the satisfaction of the requirement in question. Here, mappings (1), (3), and (5) already provide enough information for a complete argument that the table is in load position prior to the operation that causes the blanks to move from feed belt to table. Hence, we did not include mappings (2) and (4).

Finally, we note that in our methodology, we assume that safety requirements are already linked to the hazards and risks being mitigated as well as the detection method, control, and actions specified for each failure mode during Failure Mode and Effects Analysis (FMEA) [20]. We therefore focus exclusively on requirements-to-design traceability, which was the main problem based on our analysis of actual certification meetings. Of course, our approach can benefit from work that automatically synthesizes and links some of the above information to SysML models (e.g., automatic generation and linkage of FMEA reports to SysML diagrams [21]), but we do not pursue this idea further in this article.

4. Traceability Information Model

The information model in Figure 9 specifies the well-formedness criteria for the traceability links underlying our methodology in Section 3. There are three kinds of relationships in this model. (1) The structural relations between entities: These are characterized by the generalization and aggregation relations, and the association relation between `Block` and `Block Relationship` in Figure 9. (2) The traceability links that engineers manually create between entities: These are characterized by all the labelled associations in Figure 9. We refer to these links as *explicit traceability links*. (3) The links that are not explicitly created by the engineers but rather are induced by the methodology guidelines in Section 3: These are characterized by the thick dashed-line associations in Figure 9. We refer to these links as *implied traceability links*. Below, we discuss the second (explicit traceability links) and third (implied traceability links) groups of relationships.

Explicit traceability links:

- The `derive` link between System-Level Safety Requirement and its `Source`. When `Source` is a `Stakeholder`, this link specifies who has suggested a requirement. Otherwise, the link specifies what rules, policies, standards, and practices mandate a requirement (see Section 3, Step 2).
- The `derive/justify` link between `Assumption` on the system context and System-Level Safety Requirement. This link is used to capture the relation between properties of the system environment and the requirements yielded by these properties. As discussed in Section 3, Step 1, the assumptions can be formalized using SysML parametrics or OCL constraints. Recall that the system context diagram consists of blocks capturing environment entities (Environment Block) and a single block representing the system of interest (`System Block`). The latter block is further decomposed into internal system blocks during design (see Section 3, Step 4).
- The `refine` link relating System-Level Safety Requirement and the `Use Case` operationalizing that requirement (see Section 3, Step 3).
- The `decompose` link relating System-Level Safety Requirement and Block-Level Safety Requirement (see Section 3, Step 8).
- The `derive` link between Block-Level Safety Requirement and Block-Level Safety-Relevant Requirement. Recall that safety-relevant requirements are the non-safety requirements that are relevant to the fulfillment of the system's safety requirements (see Section 3, Step 2). We also use the `derive` link

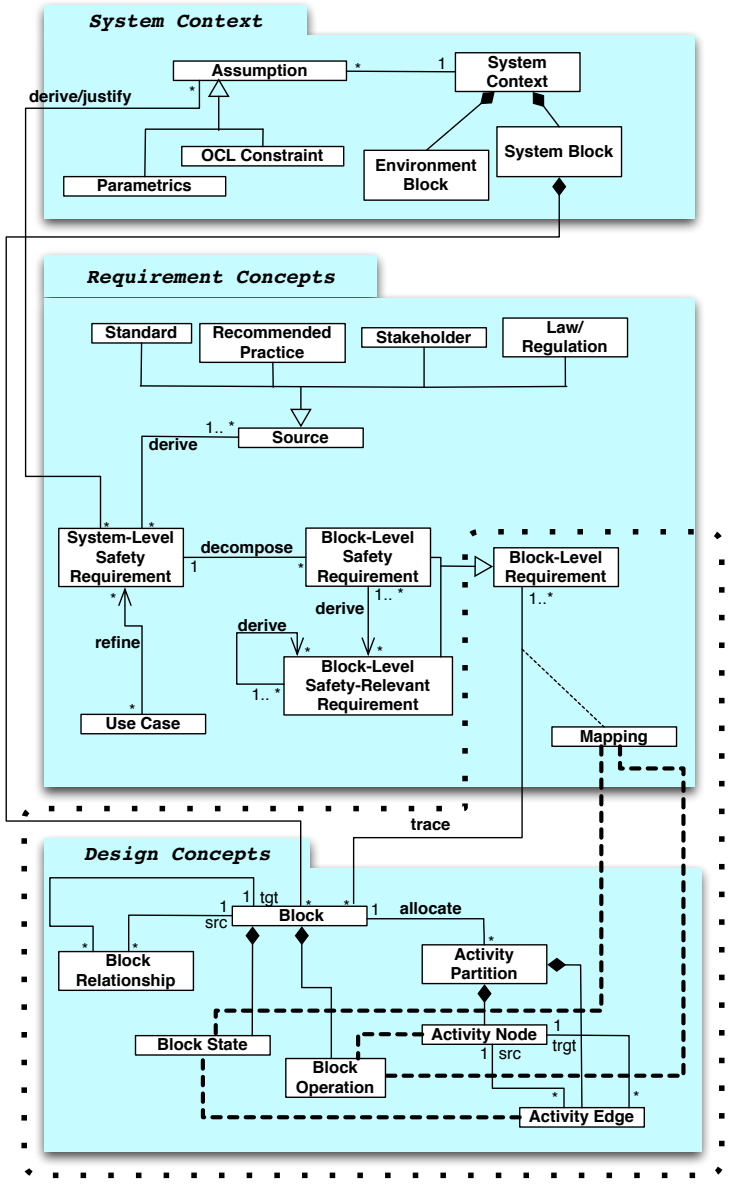

Figure 9: Traceability Information Model. The part of the model directly used by the Slicing algorithm in Section 5.2 is delineated by thick dotted line.

to specify sequences of safety-relevant requirements that directly or indirectly contribute to a particular safety requirement. This is indicated by the self-loop labelled `derive` in Figure 9.

- The `trace` link between Block-Level Requirement and `Block` to indicate the

blocks contributing to the satisfaction of those requirements (see Section 3, Step 8). As discussed in Step 9, we augment `trace` links with mappings between requirement phrases and block operations and states. This is shown through the association class `Mapping` in Figure 9.

- The `allocate` link between `Block` and design elements representing the block behavior (see Section 3, Steps 6,7). To save space, in Figure 9, we have shown only the `allocate` link for activity diagrams.

Among the links discussed above, the links labelled `derive` and `derive/justify` are new in our work. Though the rest of the links already exist in SysML, we specialize their semantics and usage to fit our application context. The `refine` link in the SysML standard is meant to be used for the same purpose as ours with the difference that we use this link to relate *System-Level Safety Requirements* to use cases, while in the standard, it is used to relate any kind of requirements to use cases [10].

The `decompose` link relates complex requirements into sub-requirements, whereas we specifically use this link to break-down system-level requirements into block-level ones. The `trace` link is a general free-form link connecting a requirement element to any requirement/model element, but we use `trace` specifically to link block-level requirements to their related blocks. We further make the `trace` links more precise using mappings. The `allocate` link is used for making connections between design elements, but the interpretation of this link is left open in SysML. In our work, we use `allocate` specifically for relating a block to the elements representing the behavior of that block. Due to lack of space we illustrate and exemplify the `trace` links only. Examples of other links are available at [15, 16].

Implied traceability links:

- As shown in Figure 9 and the example in Figure 8, the `Mapping` elements are modeled as association classes attached to `trace` links connecting block-level requirements and blocks. The content of the mappings, however, relates requirement phrases to block states and operations. The implicit relation between the content of the mappings and block states and operations is captured using the implied traceability links represented as thick dashed-line associations connecting `Mapping` to `Block Operation` and to `Block State` (see Figure 9). For example, in Figure 8, the engineer explicitly creates the `trace` links between the requirement and the `FeedBelt` and `Table` blocks, and further specifies the mapping phrases. From these mapping phrases, we can imply the traceability links between the requirement

and the operation feed_table() of FeedBelt and the state post(go_load_position()) of Table, i.e., the state where Table enters when it finishes execution of the operation go_load_position().

- A block is consistent with the activity partitions representing its behavior if: (1) For every block operation related to a safety requirement, there is at least one activity node in some activity partition related to that block. This is because every block operation related to a safety requirement describes an important behavior of that block such as an interaction between the block and other internal blocks or the environment. Since these operations describe major system behaviors, they appear in the diagrams representing block behaviors. For example, the operation go_load_position() of Table in Figure 6(a) appears in the activity partition of Table in Figure 6(b). Note that some of the block operations describing detailed behaviors may not appear as activity nodes, e.g., getter/setter operations of a block that simply retrieve/set values of some variables. (2) For every block state related to a safety requirement, there is an activity edge in some activity partition related to that block. This is because block states often refer to situations before or after execution of some block operation, and hence they can be mapped to the incoming or outgoing edges of the activity node related to that block operation. For example, post(go_load_position()), i.e., the situation where table is in load position, can be mapped to the edge from the go_load_position activity to the node for sending of the signal FeedTable in Figure 6(b).

  The consistency conditions (1) and (2) described above are respectively specified using thick dashed-lines between Block Operation and Activity Node, and between Block State and Activity Edge in Figure 9.

5. Automated Generation of Design Slices Relevant to Safety Requirements

This section explains how the traceability links described in Section 4 can be used to automatically extract slices of the design diagrams relevant to a particular safety requirement. Specifically, given a set of SysML diagrams conforming to the information model in Section 4 and given a particular block-level safety requirement r, we present an algorithm for extracting a design slice (i.e., a set of fragments of the SysML diagrams) that is relevant to r. Our algorithm refers to and uses a number of concepts in the traceability information model in Figure 9.

In Section 5.1, we formalize these concepts to define the notation used by our algorithm presented in Section 5.2. In Section 5.3, we show that our slicing algorithm is sound for temporal safety properties, and argue about the completeness of the generated slices based on our practical experience with the algorithm.

5.1. Formal Notation for the Slicing Algorithm

All the concepts that our algorithm in Section 5.2 refers to are already defined by the information model in Figure 9 in terms of classes and associations. In this section, we develop a formal (symbolic) notation for these classes and associations. Specifically, the definition below formalizes the part of the information model in Figure 9 that is used directly by our algorithm.

Definition 1 (Formalizing Traceability Information Model). *Let r be a block-level requirement. We denote the set of blocks related to r via* **trace** *links by B_r, and the set of activity partitions related to the blocks in B_r via* **allocate** *links by AP_r. The Mapping elements associated to trace links relate block-level requirements to block states and block operations. We denote the set of block states and operations related to r via* `Mapping` *elements by $Block\_St_r$ and $Block\_Op_r$. We define $Block\_Elem_r$ to be the union of $Block\_St_r$ and $Block\_Op_r$, i.e., $Block\_Elem_r = Block\_St_r \cup Block\_Op_r$. We denote the set of activity nodes related to $Block\_Op_r$ by Act_Nd_r, and the set of activity edges related to $Block\_St_r$ by Act_Ed_r. We define Act_Elem_r to be the union of Act_Nd_r and Act_Ed_r, i.e., $Act\_Elem_r = Act\_Nd_r \cup Act\_Ed_r$.*

For example, let r be the block-level requirement of Figure 8. Then, B_r is $\{$`Table`, `FeedBelt`$\}$; AP_r contains the two activity partitions in Figure 6(b); $Block\_Elem_r$ is

$\{$`feedbelt.feed_table()`, `post(table.go_load_position())`, `post(table.initialize())`$\}$

and, Act_Elem_r includes (1) the `feed_table` activity, (2) the transition from the `go_load_position` activity to the `FeedTable` signal, and (3) the transition from the `initialize` activity to the `FeedTable` signal in Figure 6(b).

Note that in $Block\_Elem_r$, `feedbelt.feed_table()` refers to a block operation, while `post(table.go_load_position())` and `post(table.initialize())` are block states. The `feed_table` activity in Act_Elem_r is related to the block operation `feedbelt.feed_table()`, the transition in Act_Elem_r from `go_load_position` to `FeedTable` is related to the block state `post(table.go_load_position())`, and the transition in Act_Elem_r from `initialize` to `FeedTable` is related to the block state `post(table.initialize())`.

Throughout the article and in our algorithm, we have referred to some concepts that are not explicitly present in the information model, but can be defined based on the existing elements in the information model. Below, we formally define these concepts.

Definition 2 (Block Diagram). *A* block diagram BD *is a tuple* $(B, BR, src_{BD}, tgt_{BD})$ *where* B *is a set of blocks,* BR *is a set of block relationships, and* $src_{BD}, tgt_{BD} : BR \rightarrow B$ *are functions respectively giving the source and the target of each block relationship.*

Definition 3 (Activity Diagram/Activity Partition). *An* activity diagram AD *is a set* AP *of* activity partitions. *Each* activity partition $ap \in AP$ *is a tuple* (N, n_0, E)*, where* N *is a finite set of activity nodes,* $n_0 \in N$ *is an initial activity node, and* $E \subseteq N \times N$ *is a set of activity edges.*

For example, the block definition diagram in Figure 6(a) has three blocks and three block relationships, and the activity diagram in Figure 6(b) has two activity partitions. As shown in Figure 9, each block can have a set of states and a set of operations, and each activity partition has a set of nodes and a set of edges. One node in each partition is designated as the initial node. Note that Definition 3 does not distinguish different types of nodes. This treatment is sufficient for the description of the algorithm in Figure 10. In Appendix B, we define a semantics for activity partitions in order to demonstrate soundness of our slicing algorithm. There, we will discuss the semantic differences between activity nodes. Particularly that activity nodes may describe one of the following: actions/activities, sending/receiving of signals or merging/splitting.

Definition 4 (Block Diagram Slice). *Let* r *be a block-level requirement. A* block diagram slice $Block\_Slice_r$ *is a block diagram* $(B_{slice_r}, BR_{slice_r}, src_{slice_r}, tgt_{slice_r})$ *where for every* $b \in B_r$*, there exists some* $b' \in B_{slice_r}$ *such that* [1]

1. $bOp \cap Block\_Op_r \subseteq bOp'$ *where* bOp *is the set of operations of* b *and* bOp' *is the set of operations of* b'*.*
2. $bAttr \cap Block\_St_r \subseteq bAttr'$ *where* $bAttr$ *is the set of attributes of* b *and* $bAttr'$ *is the set of attributes of* b'*.*

[1] Recall the notation B_r, $Block\_Op_r$, and $Block\_St_r$ from Definition 1.

Definition 5 (Activity Diagram/Partition Slice). *Let r be a block-level requirement. An* Activity diagram slice Act_Slice_r *is a set AP_{slice_r} of activity partitions where for every $ap \in AP_r$ such that $ap = (N, n_0, E)$, there exists some $ap' \in AP_{slice_r}$ such that $ap' = (N', n'_0, E')$, and* [2]

1. $N \cap Act\_Nd_r \subseteq N' \subseteq N$
2. $E \cap Act\_Ed_r \subseteq E'$
3. *There is at least one path from n_0 to n'_0 (or to all successors of n'_0) that does not go through any node in N'.*

Intuitively, Definitions 4 and 5 state that a block diagram slice related to a requirement r must include all the block operations and states directly related to r, (i.e., operations and states of the blocks in B_r intersecting with $Block\_Op_r$ and $Block\_St_r$ repectively), and similarly, an activity diagram slice related to a requirement r must include all the activity nodes and edges directly related to r, (i.e., nodes and edges of activity partitions in Act_r intersecting with Act_Nd_r and Act_Ed_r repectively). Note that these definitions are not meant to uniquely characterize a notion of design slice. Rather, they describe the minimum *syntactic* conditions (the necessary conditions) that design slices must satisfy. The slices generated by our algorithm in Section 5.2 satisfy the conditions in Definitions 4 and 5. We formally proved this in Theorem 1 in Section 5.2. In addition, our slicing algorithm preserves certain semantic properties. We discuss this in Section 5.3.

5.2. Slicing Algorithm

Figure 10 shows our algorithm for extracting block and activity diagram slices. Briefly, the algorithm identifies which model elements (i.e., activity nodes/edges, block operations/states, and relationships between blocks) can be abstracted away as they are not required for evaluation of the considered requirement. All the concepts used in the algorithm were introduced in the information model in Figure 9 and were formalized in Section 5.1. The algorithm takes as input the sets B_r, AP_r, $Block\_Elem_r$, and Act_Elem_r for a block-level requirement r. These sets are defined in Definition 1 and can be extracted from SysML diagrams conforming to the information model in Section 4. The output of the algorithm is

[2] Recall the notation AP_r, Act_Nd_r, and Act_Ed_r from Definition 1.

Algorithm. GENERATESLICE

Input: The sets B_r, AP_r, $Block\_Elem_r$, and Act_Elem_r for a block-level requirement r.
The set BR_r of block relationships between blocks in B_r, and
The functions src_r and tgt_r relating the relations in BR_r to the blocks in B_r.

Output: A block diagram slice $(B_s, BR_s, src_s, tgt_s)$, and a set AP_s of activity partition slices.

/* **Step 1. Find elements temporally related to r ($\mathbf{Design\_Elem}_r$).** */
1. $Design\_Elem_r = Block\_Elem_r \cup Act\_Elem_r$
2. **for** any block $b \in B_r$ and any element $e \in Design\_Elem_r$ **do**
3. **if** operation op of b triggers (or is triggered by) e **then**
4. $Design\_Elem_r = Design\_Elem_r \cup \{op\}$
5. **for** any activity partition $a \in AP_r$ and any element $e \in Design\_Elem_r$ **do**
6. **if** activity node n of a triggers (or is triggered by) e **then**
7. $Design\_Elem_r = Design\_Elem_r \cup \{n\}$

/* **Step 2. Extract block diagram slices ($\mathbf{Block\_Slice}_r$).** */
8. **for** every block $b \in B_r$ **do**
9. Let bOp and $bAttr$ be the sets of operations and attributes of b, respectively.
/***Remove any operation in bOp that is not in $Design\_Elem_r$.***/
10. $bOp' = bOp \cap Design\_Elem_r$
/***Remove any attribute in $bAttr$ that is not in $Design\_Elem_r$.***/
11. $bAttr' = bAttr \cap Design\_Elem_r$
12. Let bOp' and $bAttr'$ be the *new* sets of operations and attributes of b, respectively.
13. $B_s = B_s \cup \{b\}$
/* **Add block relationships and source/target functions to the block diagram slice.**/
14. $BR_s = BR_r$, $tgt_s = tgt_r$, and $src_s = src_r$

/* **Step 3. Extract activity diagram slices ($\mathbf{Act\_Slice}_r$).** */
15. **for** every activity partition $a \in AP_r$ **do**
16. Let $aNodes$ and $aEdges$ be the sets of nodes and edges of a, respectively.
17. Let $init$ be the initial node of a.
/* **Remove every node in $aNodes$ except for those in $Design\_Elem_r$, and the ending points of the activity edges in $Design\_Elem_r$.**/
18. $aNodes' = aNodes \cap \big(Design\_Elem_r \cup \{\text{the ending points of the activity edges in } Design\_Elem_r\}\big)$
/* **Remove every edge in $aEdges$ except for those whose ending points are in $aNodes'$.**/
19. $aEdges' = aEdges \cap \big(Design\_Elem_r \cup \{\text{the edges whose both ending points are in } Design\_Elem_r\}\big)$
20. Let $aNodes'$ and $aEdges'$ be the *new* sets of nodes and edges of a, respectively.
/* **Add stuttering edges.**/
21. **for** every pair $n, n' \in aNodes'$ **do**
22. **if** n' is reachable from n in a through edges none of which are in $aEdges'$ **then**
23. add a stuttering edge from n to n'
/* **Pick a new initial node.**/
24. **for** every node n in activity partition a **do**
25. **if** there is a path from $init$ to n that does not go through any node in $aNodes'$ **then**
26. mark n as a *new* initial node of a.
27. **if** more than one node is marked as initial node **then**
28. add a new initial to a with transitions to the old ones.
29. $AP_s = AP_s \cup \{a\}$

Figure 10: Algorithm for generating design slices.

a block diagram slice (Definition 4) and an activity diagram slice (Definition 5). The algorithm has three main steps discussed and exemplified below.

Step 1 (Find $\mathbf{Design\_Elem}_r$). This step identifies the design elements that are temporally related to r. The set $Design\_Elem_r$ is initially set to include the block and activity diagram elements that are directly related to r via the explicit and implied traceability links suggested by our information model (Figure 9). We then compute the set of block operations and activity nodes that trigger (or are triggered by) the elements in $Design\_Elem_r$. We do so by adding to $Design\_Elem_r$ any block operation or activity node that triggers (or is triggered by) an existing element in $Design\_Elem_r$. The resulting $Design\_Elem_r$ is the set of design elements that are temporally related to r. Recall that in our methodology (Steps 6 and 7 in Section 3), we discussed how the temporal relationships between block operations and activity nodes can be identified to compute $Design\_Elem_r$ (see [15] for more details). If the behavioral diagrams do not fully comply with our methodology, we can still build $Design\_Elem_r$ using existing techniques for control dependence analysis [22].

For example, $Design\_Elem_r$ for the requirement in Figure 8 is initially set to the union of $Block\_Elem_r$ and Act_Elem_r which includes the following elements:

- the block operation `feedbelt.feed_table()`
- the block state `post(table.go_load_position())`
- the block state `post(table.initialize())`
- the activity node `feed_table`
- the activity transition from `go_load_position` to `FeedTable`
- the activity transition from `initialize` to `FeedTable`

After executing Step 1, $Design\_Elem_r$ would be extended to include the following elements in addition to the above ones:

- the block operation `feedbelt.go_load_position()`
- the block operation `feedbelt.initialize()`
- the activity node related to receiving signal `FeedTable` in the activity partition related to `FeedBelt` in Figure 6(b)
- the activity node related to sending signal `Go_Unload_Position` in the activity partition related to `FeedBelt` in Figure 6(b)

The block operations `feedbelt.go_load_position()` and `feedbelt.initialize()` are added because they trigger the block states `post(table.go_load_position())` and `post(table.initialize())`, respectively. The two activity nodes in the above list are added because the former triggers the activity node `feed_table`, while the latter is triggered by the same activity node.

Step 2 (Extract $\mathbf{Block\_Slice}_r$). This step abstracts away attributes and operations not present in $Design\_Elem_r$ from blocks in B_r. For example, the block diagram slice related to the requirement in Figure 8 is shown in Figure 11(a).

Step 3 (Extract $\mathbf{Act\_Slice}_r$). This step abstracts away every activity node from activity partitions in AP_r that is not in $Design\_Elem_r$ or is not an ending point of an edge in $Design\_Elem_r$. It also removes every edge that is not in $Design\_Elem_r$. To maintain connectivity between the nodes, after the removal of edges, the last part of the algorithm adds special edges between those nodes whose connecting paths are removed. These edges are meant to preserve only the reachability relations between nodes and not the exact number of steps to go from one node to another. For this reason, we call them *stuttering edges* [23]. For example, the activity slice related to the requirement in Figure 8 is shown in Figure 11(b). The stuttering transitions between activity nodes are shown as dashed arrows. After adding the stuttering transitions, for each activity partition, we identify an initial node. To do so, we find a node to which there is a path from the initial node of the original non-sliced activity partition that does not go through any other node in the sliced activity partition. If several nodes in the activity partition satisfy this condition, we create a dummy initial node with transitions to all of these nodes.

Theorem 1. *Our algorithm in Figure 10 generates block diagram and activity diagram slices, (i.e., the block diagram and activity diagram slices generated by the algorithm in Figure 10 satisfy the conditions in Definitions 4 and 5, respectively.).*

The proof of the above theorem is given in Appendix A. Adding stuttering transitions and identifying initial nodes of activity diagram slices allow us to reason about the soundness of our algorithm (see the discussion on soundness in Section 5.3). Note that the notation for activity diagram slices is slightly different from the conventional SysML/UML activity diagram notation mainly due to addition of stuttering transitions. However, this notational difference does not hinder the use of existing standard SysML/UML tools for manipulating the slices because we can develop a profile to extend the SysML notation to include stuttering transitions, and hence, use such tools to create diagram slices.

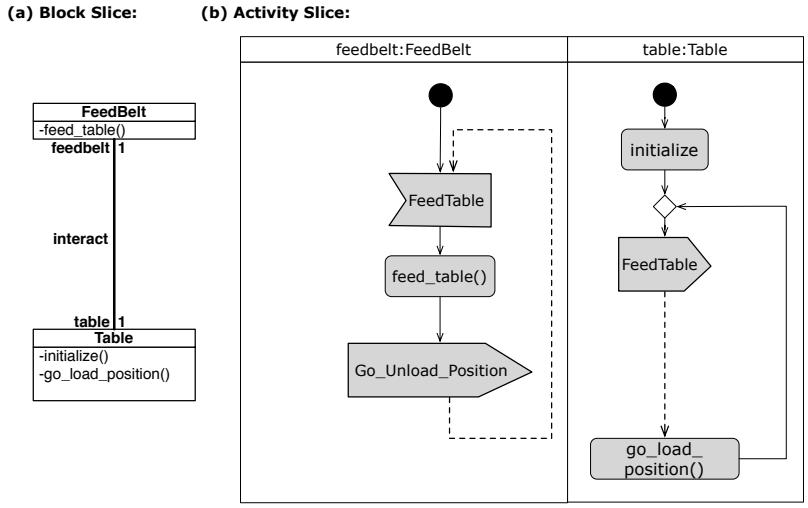

Figure 11: The block and activity slices for the requirement in Figure 8 extracted from the SysML design diagrams in Figure 6.

5.3. Properties of Design Slices

Ideally, the design slices generated by our algorithm should possess the following two properties in order to be effectively used by certifiers for verifying safety requirements.

Soundness If a requirement holds over a design slice, it should also hold over the original (non-sliced) design.

Completeness If a requirement holds over the original (non-sliced) design, then the design slice related to that requirement should contain enough information to conclusively verify that requirement.

The first property (soundness) ensures that our algorithm generates correct design slices. If not sound, the generated design slices cannot be trusted because a requirement may hold over a design slice, while the original design does not satisfy it. The second property (completeness) ensures that the certifier can always rely on checking the generated design slices and never need to refer to the original design. If not complete, then there are requirements for which analysis of the design slices does not yield a conclusive result, while the original design contains sufficient information for decisive verification or refutation of those requirements. Between the above two properties, soundness is a more crucial one. Obviously, if the algorithm is unsound, it cannot be used in certification. Failure to satisfy the

completeness property, however, does not make the algorithm inapplicable, but requires the certifier to refer back to the original design whenever the analysis of slices is inconclusive. Below, we discuss soundness and completeness properties of our algorithm.

Soundness. As we explained already, the activity diagram slices are created in such a way that the reachability relations between the nodes in the original activity diagrams are preserved in the slices. This enables us to keep the temporal orderings of the nodes in the slices consistent with those of the nodes in the original diagrams, and hence, ensure that the slices are sound for requirements expressible as temporal constraints. Note that many safety properties are indeed temporal constraints because they often state in what order the actions should occur so that the system does not end up in an unsafe or undesirable state [24]. For example, the requirement in Figure 8 is a temporal constraint, requiring `go_load_position`() or `initialize`() to occur before `feed_table`(), and hence ensuring that table is in the desired position prior to the execution of `feed_table`(). The slice in Figure 11(b) is sound for analyzing the requirement in Figure 8. This is because the orderings between sending of signal `FeedTable` and `go_load_position` and `initialize` activities, and between receiving of the `FeedTable` signal and the `feed_table` activity in the activity diagram slice in Figure 11(b) are the same as the orderings between these nodes in Figure 6(b), In Appendix B, we formally prove that our slicing algorithm in Figure 10 generates activity diagram slices that are sound for verifying temporal safety requirements.

Completeness. As mentioned above, completeness is a less crucial property than soundness. Automated techniques are often partially complete. In our work, it is difficult to demonstrate that the generated design slices always contain sufficient information for analyzing safety requirements because: First, completeness of a generated design slice depends on the completeness of the traceability links and mappings attached to the traceability links. For example, if we remove from Figure 8 either of the mappings related to `post`(`table.go_load_position`()) or `post`(`table.initialilize`()), the resulting activity partition slices in Figure 11(b) will not include the activity nodes `go_load_position` and `initialize` respectively. Second, the ability of the certifier to analyze the design depends on several factors, in particular, their background on the language used for the design and their knowledge of the domain under analysis. As a result, different people may require different amounts of information to verify certain requirements. However, due to the subjectiveness of this issue, we plan to evaluate complete-

ness of our slicing algorithm using empirical techniques by running controlled experiments. However, we expect our slicing algorithm to be complete for a large number of safety requirements. In particular, our analysis has shown that our algorithm is complete for all of the safety requirements in our case studies described in Section 7 when sufficient traceability links and sufficient mapping elements are provided.

For example, we can argue that the block and activity diagram slices in Figure 11 contains enough information to check the requirement (r) in Figure 8. To check r, we need to demonstrate that (1) in the block diagram slice, there is an association relation between the blocks referred to by r, and (2) the sequence of interactions in the activity diagram slice satisfies r. The block diagram slice in Figure 11(a) fulfills the former condition. To show the latter, we need to show that $p_1 \wedge \neg p_2$ never happens in the design (see Figure 8 for p_1 and p_2). In this example, this translates into showing that `feed_table` of `FeedBelt` cannot occur unless either `go_load_position` or `initialize` of `Table` has already happened. The activity slice in Figure 11(b) shows this is the case, i.e., `feed_table` can only occur when it has received the signal `FeedTable`. This signal is sent only after `go_load_position` or `initialize` is executed. Note that the stuttering transitions between sending of `FeedTable` signal and `go_load_position` activity indicates that the `go_load_position` activity does not necessarily occur immediately after sending of `FeedTable` as this edge abstracts several steps that perhaps may involve receiving of several signals from the environment. But there is no delay during the execution of normal activity diagram transitions, i.e., the `feed_table` activity occurs immediately after receipt of the `FeedTable` signal. Based on this discussion, it can be seen that the slices in Figure 11 are complete for analyzing the requirement in Figure 8.

6. Tool Support

We have developed a tool named SafeSlice (`http://modelme.simula.no/pub/pub.html#ToolSlice`) in support of our approach. Specifically, SafeSlice enables users to: (1) specify the traceability links envisaged by the traceability information model described in Section 4; (2) check the consistency of the established links; (3) automatically extract slices of design with respect to requirements using the slicing algorithm in Section 5; (4) use slices for conducting inspections and ensuring that the design satisfies the safety requirements; and (5) generate reports. The reports can be customized to include various types of information, e.g.,

design slices, whole designs, statistics, and pie charts for progress monitoring of inspections.

To facilitate the application of our tool in real settings and for the tool to be more easily maintainable, we built the tool as a plugin for a major modeling environment, called Enterprise Architect (`http://www.sparxsystems.com.au`). In addition to providing mature facilities for SysML modeling, Enterprise Architect offers built-in support for managing additional information related to a modeling project. In our case, we used this capability for managing the information related to design inspections.

SafeSlice builds on Microsoft ActiveX COM technology. We used Microsoft .NET Framework 2.0 and Visual Studio 2008 as the development platform. SafeSlice is written in Visual C# and is roughly 10,000 lines of code excluding comments and third-party libraries.

Checking compliance of the links to the traceability information model in Section 4 and extraction of slices in SafeSlice require a negligible amount of execution time. We recorded the time required to check information model compliance and produce design slices. For the larger of the two systems in our studies in Section 7 (i.e., PCS), SafeSlice took about half a minute to check the compliance of the entire set of traceability links, and a maximum time of ten seconds to produce a design slice.

7. Case Studies and Lessons Learned

To validate the feasibility and effectiveness of our approach, we have conducted two case studies: the first case study is the Production Cell System, a small fragment of which was introduced in Section 3 as the running example for this article. The full case study material is available at [15]. The second case study is a real-world industrial system from the maritime and energy domain. For further details about this case study, consult [7]. Below, we first describe the two case studies (Section 7.1), and then report on the experience gained (Section 7.2).

7.1. Description of the Case Studies

Context. In our first case study, we applied our methodology to the Production Cell System (PCS). As discussed earlier in Section 3, the aim of PCS is the transformation of metal blanks into forged plates (by means of a press) and their transportation from a feed belt into a container. PCS is composed of six devices that are applied to the blanks fed into the system in a specified order. Each of the devices

is equipped with sensors and actuators, enabling them to operate and communicate with one another. The design of PCS must meet several safety requirements some of which were earlier discussed in Section 3. The PCS case study serves as an initial validation of our methodology and was conducted before applying the methodology to a real industrial system (the second case study).

Our second case study concerns a safety critical IO module developed at a maritime and energy company specializing in computerized systems designed 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. The role of the IO modules in these systems is to connect software control components to hardware and mechanical devices. The IO module in our study was designed to transfer specific commands from a remote control unit to a fire detection panel. The panel was intended for marine applications (e.g., general cargo and passenger vessels, and offshore installations) and could be set up to control a variety of fire detection sensors.

Case Study Selection. The choice of PCS as a case study was driven by two main considerations: First, PCS is a well-known exemplar for embedded systems [12]. In PCS, similar to many safety critical systems in domains such as avionics, automotive, maritime and energy, the software control modules are integrated with hardware devices and continuously interact with the environment. A second advantage of PCS is that it has been already analyzed using several existing formal and semi-formal approaches [12], thus providing us with a large source of useful information about the system. Most notably, we benefitted from an earlier object-oriented design of PCS built using the Fusion method [25]. This earlier design is based on a real production cell and also contains a complete list of safety requirements. In lieu of access to the real system, this design served as an important aid both to better understand the system and further to compare alternative design decisions.

With regards to our industrial case study, our main selection criterion was that the chosen IO module should be representative. The structure and behavior of the module that we selected for our case study was deemed by the lead engineer of the IO modules as being representative of the significant majority of the IO modules developed at the company.

Results. For the first case study, we developed a complete SysML model of the PCS with traceability to the PCS safety requirements. The model includes all the SysML diagrams in our methodology. In particular, it consists of 58 diagrams, 479

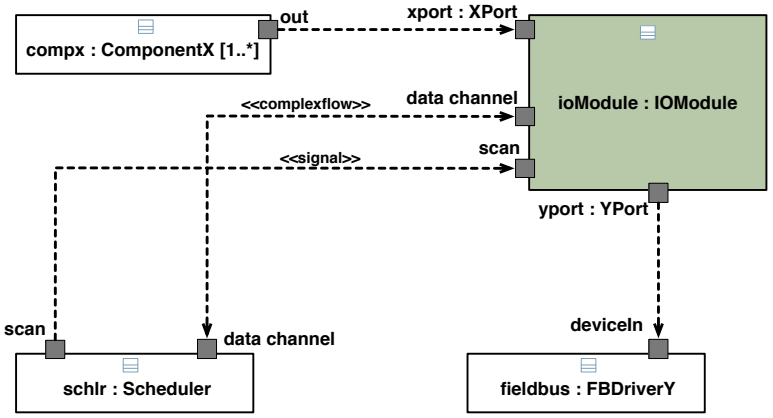

Figure 12: Architectural links for the IO module in the industrial case study expressed as a SysML IBD.

elements having 419 relations and 189 attributes. We generated 10 design slices for 10 safety requirements in the description of PCS using our slicing algorithm. Excerpts of some of the key artifacts in the PCS case study and slices thereof were used for illustration in earlier sections of the article (see Figures 3–8 and Figure 11).

In the second case study, we applied our methodology to the IO module under investigation and developed a complete SysML model with traceability to the module's requirements. To understand the module, we relied primarily on interviews with the developers, and an analysis of the existing documentation and source code. The resulting SysML model (including the traceability links) was iteratively validated and refined in collaboration with the lead engineer of the IO modules.

For example, Figure 12 shows the (sanitized) IBD developed for the IO module to capture its communications with the control modules (instances of ComponentX), the scheduler, and a lower-level real-time driver (instance of FBDriverY). Figure 13 shows the activity diagram representing the overall behaviour of the IO module, and in particular, its main internal threads. The IO modules developed by the company use a complex multithreading structure to increase performance and to allow for more configurability. Capturing the multithreaded structure (both at the level shown in Figure 13 and at lower levels) was one of the main modeling complexities in this case study. Lastly, Figure 14 shows a (sanitized) BDD representing the internal structure of IO module.

Since the IO modules at the partner company are deployed in safety monitoring and control systems, most of the modules' requirements are safety-relevant.

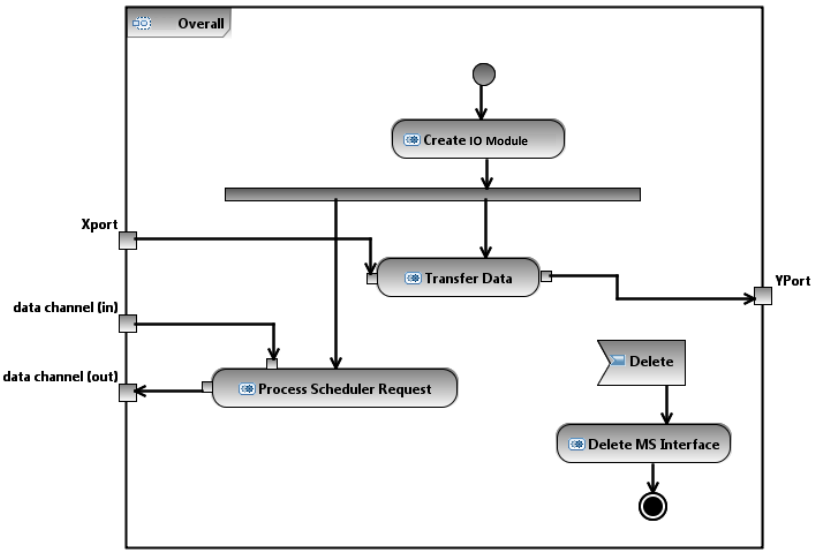

Figure 13: Overall behaviour of the IO module in the industrial case study expressed as a SysML AD.

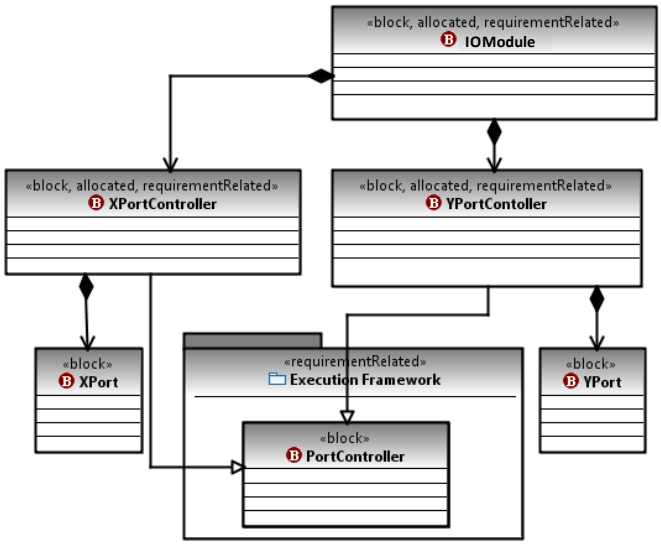

Figure 14: Structure of the IO module in the industrial case study expressed as a SysML BDD.

As we stated earlier, this means that the requirements of these modules in some way contribute to the satisfaction of the system-level safety goals. An example

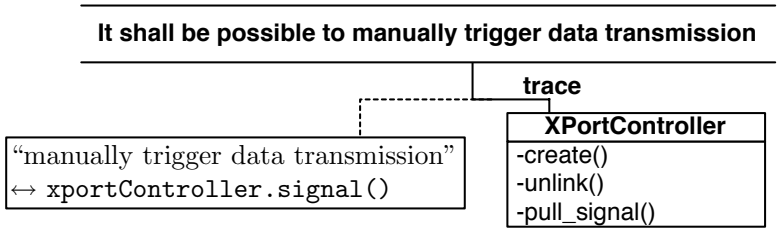

Figure 15: An example requirement from the IO module in the industrial case study with traceability links to the design.

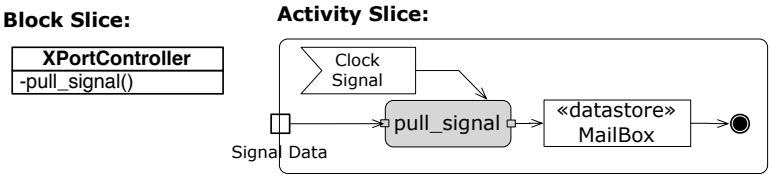

Figure 16: Design slice for the requirement in Figure 15.

safety-relevant requirement from our case study IO module is *“It shall be possible to manually trigger data transmission”*. This requirement is shown in Figure 15 along with traceability links from it to the design. The design slice that is generated automatically for this requirement by our slicing algorithm is shown in Figure 16.

The SysML design developed in our industrial case study includes all the SysML diagrams envisaged by our methodology. Specifically, the design consists of 23 diagrams, 194 elements having 186 relations and 57 attributes. The IO module under study included 30 safety-relevant requirements, all of which where traced to the SysML design through appropriate traceability links. Subsequently, design slices were generated automatically for all these requirements.

7.2. Lessons Learned

In this section, we discuss the lessons learned from applying our methodology to the two case studies described in Section 7.1.

Prerequisite: Use of SysML. The applicability of our methodology depends on how feasible it is to use SysML in industrial contexts. Overall, we found SysML to be a good fit for capturing the behavioral and structural characteristics of the systems in our studies. We did not encounter challenges that would indicate an inadequacy in the expressive power of the SysML language for system design, nor did we come across areas where using SysML

made the design more complex than necessary (i.e., accidental complexity). In comparison to the UML language, we found two aspects of the SysML language to be advantageous for systems engineering. Firstly, SysML can be used for capturing *both* object-oriented and non-object-oriented systems, whereas UML is aimed at only the former type of systems. The ability to handle non-object-oriented systems is particularly important for embedded control systems, because a significant proportion of these systems, including the IO module in our industrial case study, are not object-oriented. In our case studies, we applied the same methodology with an equal degree of success for capturing both object-oriented (PCS) and non-object-oriented systems (IO module). A second main advantage of SysML is the introduction of parametric diagrams, an example of which was shown in Figure 4. We found parametric diagrams very useful and a natural mechanism for specifying the operating environment for software in the presence of electrical and mechanical parts.

Costs: Level of Required Effort. The effort required to establish traceability links according to our methodology was manageable. In our industrial case study, the design and tracing activities took about three weeks, involving approximately 40 man-hours of effort. In the PCS case study, these activities took about five weeks, involving approximately 90 man-hours of effort. In both cases (the former being also confirmed by our industrial partner), the required effort seems acceptable, considering that: 1) such systems have a long lifetime, 2) the use of SysML include additional benefits like reuse, standard compliance, and reduced ambiguity and inconsistency, and finally, 3) the methodology is intrinsically iterative and the level of details to model is decided by the designer according to time and schedule availability.

Effectiveness: Reduction of the Size of Design Slices. The slicing procedure reduced the size of the design models that needed to be reviewed in order to determine whether the design satisfied a requirement of interest. The size of design models is computed as the total number of elements in the models, e.g., the elements in the design concepts package in Figure 9. The average reduction rate was 98% both over the PCS requirements and over the IO module requirements, thus giving an average reduction rate of 98% for the combined set of requirements from both systems. Figure 17 shows the reduction frequency distribution and quantile box plot for the combined set of requirements. As indicated by the distribution, the variation range is very

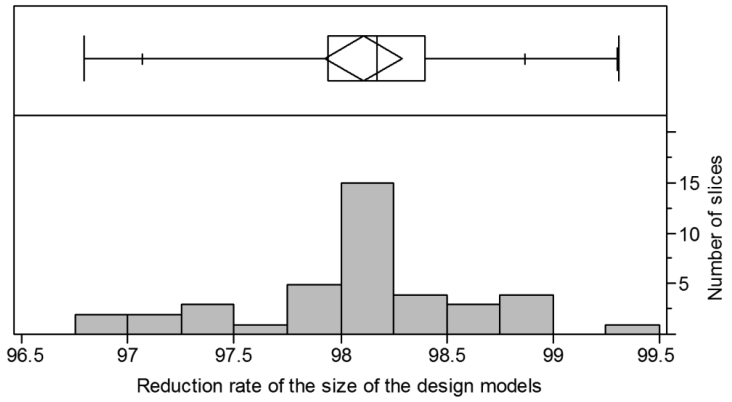

Figure 17: Frequency distribution and quantile box for slicing reduction

small, considering that the two systems are in different domains, and were modeled by different people. We therefore anticipate reduction rates in size to be close to the values observed here for other systems modeled according to our methodology. It is reasonable to assume that inspection effort is related to the size of the models that are required to be inspected.

In Section 5.3, we provided a formal proof of soundness for our slicing algorithm, showing that the temporal sequencing of activities is preserved from the source (activity) models to the sliced models. This means that the slices never provide misleading information to the inspectors about the ordering of activities. As to whether a slice provides complete information to conduct the task the slice is intended for, as we already argued in Section 5.3, one cannot give a formal proof, unless one enforces major restrictions on the requirements and design specifications, and introduces more sophisticated formal methods which could in turn reduce the applicability of our methodology. In the two case studies we performed, we observed that the sliced models provided adequate information for performing the task at hand (namely, checking if a given requirement is properly realized by the design). This was mainly attributable to the fact that we had high-quality traceability links from the requirements to the design, thus mitigating the possibility that slicing would filter out too much information. In general, if a certain piece of information is deemed missing from a slice, the inspectors can always access the original model. Further, they can update the traceability links so as to ensure that the missing information will be included in the slices that will be generated in the future.

8. Related Work

Our work is motivated by the need to improve the accuracy and efficiency of design inspections during safety certification. Several approaches already exist in the literature that targeted at improving software inspections. Particularly, Bazzana et al. [26] propose a methodological basis for software assessment and a supporting tool environment, noting certification as one of the area where their approach could be applied. Juristo and Morant [27] propose a framework for validation and verification (V&V) of both knowledge-based and conventional software, with inspections as one of the main parts of their framework. Kelly and Shepard [28] describe an inspection technique and a process adopted to introduce inspection in an industrial environment. Our work differs from the above work in two ways: First, the scope of our work is restricted to system design specifications rather than software development artifacts in general. This enabled us to provide more specific guidance tailored to the specific concerns in system design. A second but related difference between our work and the above work is that our guidelines are not aimed at improving the inspection processes per se, but rather on how to "build" designs that can be more accurately and efficiently inspected using the (current) inspection procedures in software safety certification.

A major aspect of our work in this article was establishing traceability between safety requirements and design. Existing work on traceability primarily addresses the question of how to automatically discover the traceability links. Various techniques have been proposed, among many others: Cleland-Huang et al. [29] apply information retrieval techniques to find candidate links between development artifacts; Egyed [30] utilizes run-time information to suggest links between the system's models and the system's implementation; and Jirapanthong and Zisman [31] provide a rule-based approach for inferring links between product-line artifacts. These techniques are all applicable for systems that are subject to safety certification. However, they must be viewed as complementary to, and not a replacement for, methodologies that help developers manually create complete and correct links at the right level of detail and thus mitigate the risk of a very lengthy certification process whose cost would dwarf the overhead associated with manual traceability links. Further, safety-critical software is often several folds more expensive to build than non-safety-critical software; hence, the traceability overhead makes up for a much smaller part of the total development cost.

Using a traceability information model to systematize the construction of traceability links is not new. Generic information models already exist for characterizing the links for various development tasks. For example, Ramesh and Jarke

[32] provide such an information model based on an observation of the practices in several software organizations, and Panesar et al. [33] – based on an analysis of the traceability criteria in the IEC 61508 standard. These information models aim to be independent from the development artifacts and hence cannot specify either the detailed structure of the traceability links or the methodology that must be followed for establishing them. In contrast, in our work, we assume that the design artifacts are expressed using SysML models, thus enabling us to elaborate the structure of the links and provide a concrete methodology for creating them.

More recently, there has been a growing interest in traceability information models for MDE [34], the insight being that such models must be built for the specific needs of a problem, and the notations used throughout development. Our work in this article applies this general idea to develop an information model for the specific needs of "unrestricted natural language requirements", "software safety certification" and the "SysML" notation.

The motivations for establishing traceability between development artifacts go beyond what we considered in this article. One notable topic outside the scope of our current work but intimately related to traceability is change impact analysis. Change impact analysis is concerned with "identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change" [35]. Various techniques exist for change impact analysis, with often differing requirements about the traceability links that need to be defined and also the semantics of the links. For example, Briand et al. [36] propose a traceability information model and an algorithm based on this information model for automatically analyzing the impacts of change in UML models. ten Hove et al. [37] develop and use a first-order logic formalization of requirements relations for change impact analysis in SysML requirements models. While useful, the conceptualization of the traceability links in these approaches is at a coarser level of abstraction that what is necessary for safety certification, as the links were envisaged for different analysis purposes.

Related to slicing, several techniques already exist with the aim of reducing complexity in various development tasks. Most of the existing work on slicing is concerned with program code, where slicing is mainly used as a debugging aid [38]. Other applications for code slicing have also been noted in the literature including program comprehension, software maintenance, and testing (see [39, 40] for surveys). For models, slicing has been studied primarily as a way to reducing cognitive load and to improve understanding, inspection and modification of models. Various model slicing techniques have been proposed, e.g., Korel et al. [41] provide a technique for the slicing of state-based models using dependence anal-

ysis, and Kagdi et al. [42] a technique for slicing UML class models based on predicates defined over the model's features. These approaches, in contrast to ours, are not aimed at expressing the relationships between the requirements and the design, and hence cannot be used for extracting design slices with respect to a given requirement.

MDE languages such as UML and SysML are extensible by design. Indeed, it is now increasingly common to develop standard extensions to these languages to meet domain- or analysis-specific needs. For example, Soares et al. [43] extend SysML requirements diagrams with new stereotypes for classification of requirements into functional, non-functional, and external. They also propose additional properties for requirements, e.g., priority to facilitate requirements analysis. The main factor that distinguishes our work from the above work and other SysML extensions lies in the ends (i.e., purpose) to be achieved by the extension. To our knowledge, there is no prior SysML extension specifically designed to support the needs of design safety inspections. Addressing this gap constitutes one of the main contributions of our work.

Our concept of design slice is similar in theme and aim to architectural views. In fact, both slices and architectural views capture a fragment of the whole design, consisting of several diagram types (e.g., class diagrams and activity diagrams). In both cases, the fragment is extracted with the aim of reducing complexity and the effort needed for analysis and review. However, in the context of software architecture, the design is typically at a higher level of abstraction than in our context. Moreover, a design slice is related to a given (safety) requirement whereas an architectural view is related to a set of concerns which, in contrast to our work, represent higher-level goals, including non technical aspects such as social, psychological, and managerial issues [44, 45, 46].

9. Conclusion

In this article, we developed a framework to facilitate software design inspections conducted as part of the safety certification process. Our framework is grounded on SysML which is rapidly becoming the notation of choice for developing safety-critical systems. The framework includes a traceability information model, a methodology to establish traceability, and mechanisms to use traceability for extracting slices of models relevant to a particular (behavioral) safety requirement. Our slicing algorithm enables certifiers and safety engineers to narrow the scope of their analysis to the small fragments of the design related to the task at hand. This helps reduce cognitive load and thus makes it less likely that seri-

ous safety issues would be overlooked. We have validated our approach on one benchmark and one industrial case study. Lastly, we have developed tool support for our methodology, implemented via plug-ins in a major SysML modeling environment.

Our methodology exploits and enhances SysML capabilities for system design and for capturing traceability between requirements and design. Generalizability to other notations has not been our main intent. Instead, our focus has been on ensuring that both the design and the links between the requirements and the design are ”precise” enough to allow for systematic design inspections. Achieving precision required grounding our work on a specific notation. We chose SysML to base our work on due to its rising popularity in the embedded systems domain. That said, some theoretical aspects of our work are indeed generalizable. For example, our slicing algorithm can be generalized to other notations if the sets of input elements and output elements can be adapted for those notations. Note that the soundness results in our paper can also easily be reused for notations with operational semantics.

In this paper, we focused on behavioral safety and safety-relevant requirements because: (1) Certifiers mainly focus on safety and safety-relevant requirements during inspections, and (2) non-behavioral requirements cannot typically be traced to particular design fragments, and hence, our work is not applicable to them. For example, performance requirements are typical non-behavioral requirements that can affect system safety goals. Since these requirements constrain the behavior of the system as a whole, they cannot be inspected by focusing on a particular design fragment.

The work reported in this article matches a major industrial need. On the one hand, the industry has long recognized the value of model-based engineering for managing complexity, but on the other hand, there is not an adequate level of support, in terms of methods and tools, for building and relating models in a way that is suitable for the analyses that need to be performed over the models, e.g., certification, impact analysis, automated testing and verification. As a result, developers often do not model the right aspects of the system or miss important details. This problem must be addressed through the development of more concrete and validated methodologies and tools. Our work here was a step towards this goal, focused on safety certification. Although there is certainly a cost overhead for using our proposed framework, we believe the overhead is justified given the overall development costs and the formidable cost and schedule risks that poor traceability can pose during certification. Further, our framework targets (behavioral) safety requirements. Hence, traceability costs are driven by the extent of

safety-relevant aspects, not the size of the entire system. These aspects are typically small size-wise, but need very careful analysis.

This article is directed towards inspections, which constitute only one class of the several quality assurance measures that can be taken during the development process. Inspections alone are seldom enough for ensuring the quality of the design. Other important quality assurance measures taken in tandem with and complementing inspections include prototyping, simulation, animation, and verification. In the future, we plan to investigate how we can expand our SysML-based methodology to support other types of quality assurance measures. Particularly, we would like to study how SysML models can be used for simulation and automated design verification. Another thread for future work is the development of a more structured framework for expressing requirements. Two promising directions that we plan to investigate related to requirements specification are: (1) natural language requirement templates [47] and (2) requirement annotations [48]. Having a more rigorous requirements specification framework will not only improve requirements quality and reuse, but will also lead to a more precise and flexible mapping of the requirements onto design elements (e.g., block operations and states). This will in turn help with the extraction of better design slices and also will pave the way for extending our work to non-behavioral safety requirements. Lastly, we plan to conduct a more thorough evaluation of our work through controlled experiments and larger industrial case studies. The main goal of the evaluation will be to assess the extent to which developers benefit from our framework and thus obtain a more conclusive picture of the cost-benefit tradeoffs for traceability in the context of safety certification. As a first step, we have been quantitatively evaluating the usefulness of our slicing algorithm through a controlled experiment with promising results. For details, see [49].

References

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

[2] DO-178B - software considerations in airborne systems and equipment certification, Radio Technical Commission for Aeronautics (RTCA) Inc (1992).

[3] Road vehicles – functional safety, ISO draft standard (2009).

[4] J. Holt, S. Perry, SysML for systems engineering: Institute of engineering and technology (2008).

[5] International Council on Systems Engineering, http://www.incose.org/.

[6] D. Falessi, S. Nejati, M. Sabetzadeh, L. Briand, A. Messina, Safeslice: a model slicing and design safety inspection tool for sysml, in: SIGSOFT FSE, 2011, pp. 460–463.

[7] M. Sabetzadeh, S. Nejati, L. Briand, A. Evensen Mills, Using SysML for modeling of safety-critical software-hardware interfaces: Guidelines and industry experience, in: HASE, 2011, pp. 193–201.

[8] W. Schafer, H. Wehrheim, The challenges of building advanced mechatronic systems, in: FOSE '07, 2007, pp. 72–84.

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

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

[11] Survey of model-based systems engineering (MBSE) methodologies, INCOSE Survey (2008).

[12] C. Lewerentz, T. Lindner (Eds.), Formal Development of Reactive Systems - Case Study Production Cell, Vol. 891 of LNCS, Springer, 1995.

[13] H. Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley, 2000.

[14] B. Bruegge, A. Dutoit, Object-oriented software engineering - using UML, patterns and Java (3rd ed.), Prentice Hall, 2009.

[15] L. Briand, T. Coq, T. Klykken, S. Nejati, R. Panesar-Walawege, M.Sabetzadeh., Using SysML to support safety certification: A methodology and case study, Tech. Rep. 2, SRL-DNV, 92 pages, Available at: http://vefur.simula.no/~shiva/report2.pdf (December 2009).

[16] T. Klykken, A case study using SysML for safety-critical systems, Ph.D. thesis, University of Oslo, available at: http://vefur.simula.no/~shiva/tonje.pdf (2009).

[17] D. Jackson, M. Thomas, Software for Dependable Systems: Sufficient Evidence?, National Academy Press, 2007.

[18] A. van Lamsweerde, Requirements Engineering: From System Goals to UML Models to Software Specifications, Wiley, 2009.

[19] D. Harel, R. Marelly, Specifying and executing behavioral requirements: the play-in/play-out approach, Software and System Modeling 2 (2) (2003) 82–107.

[20] C. Ericson, Hazard Analysis Techniques for System Safety, JOHN WILEY & SONS, 2005.

[21] P. David, V. Idasiak, F. Kratz, Reliability study of complex physical systems using SysML, Reliability Engineering & System Safety 95 (4) (2010) 431 – 450.

[22] K. Androutsopoulos, D. Clark, M. Harman, Z. Li, L. Tratt, Control dependence for extended finite state machines, in: FASE, 2009, pp. 216–230.

[23] M. Abadi, L. Lamport, The existence of refinement mappings, in: LICS, 1988, pp. 165–175.

[24] E. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 1999.

[25] S. Barbey, C. Peraire, D. Buchs, A case study for testing object-oriented software: A production cell (1998).

[26] G. Bazzana, R. Brigliadori, R. Cole, K. Kirkwood, F. Seigneur, Techniques and tools for software assessment and certification, Annual Review in Automatic Programming 16 (Part 2) (1992) 153 – 160.

[27] N. Juristo Juzgado, J. Morant, Common framework for the evaluation process of kbs and conventional software, Knowl.-Based Syst. 11 (2) (1998) 145–159.

[28] D. Kelly, T. Shepard, Task-directed software inspection, Journal of Systems and Software 73 (2004) 361–368.

[29] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, E. Romanova, Best practices for automated traceability, IEEE Computer 40 (6) (2007) 27–35.

[30] A. Egyed, A scenario-driven approach to traceability, in: ICSE, 2001, pp. 123–132.

[31] W. Jirapanthong, A. Zisman, XTraQue: traceability for product line systems, Software and System Modeling 8 (1) (2009) 117–144.

[32] B. Ramesh, M. Jarke, Toward reference models for requirements traceability, IEEE TSE 27 (1) (2001) 58–93.

[33] R. K. Panesar-Walawege, M. Sabetzadeh, L. Briand, T. Coq, Characterizing the chain of evidence for software safety cases: A conceptual model based on the IEC 61508 standard, in: ICST, 2010, pp. 335–344.

[34] P. Mader, O. Gotel, I. Philippow, Getting back to basics: Promoting the use of a traceability information model in practice, in: IEEE TEFSE '09: ICSE09 Wrkshp, 2009, pp. 21–25.

[35] S. Bohner, R. Arnold, Software Change Impact Analysis, IEEE Computer Society, 1996.

[36] L. Briand, Y. Labiche, T. Yue, Automated traceability analysis for uml model refinements, Information & Software Technology 51 (2) (2009) 512–527.

[37] D. ten Hove, A. Göknil, I. Kurtev, K. van den Berg, K. de Goede, Change impact analysis for sysml requirements models based on semantics of trace relations, in: Proceedings of the ECMDA Traceability Workshop (ECMDA-TW), 2009, pp. 17–28.

[38] M. Weiser, Program slicing, in: Proceedings of the 5th International Conference on Software Engineering (ICSE'81), 1981, pp. 439–449.

[39] F. Tip, A survey of program slicing techniques., Tech. rep., Amsterdam, The Netherlands (1994).

[40] D. Binkley, K. Gallagher, Program slicing, Advances in Computers (1996) 1–50.

[41] B. Korel, I. Singh, L. Ho Tahat, B. Vaysburg, Slicing of state-based models, in: 19th International Conference on Software Maintenance (ICSM'03), 2003, pp. 34–43.

[42] H. Kagdi, J. Maletic, A. Sutton, Context-free slicing of uml class models, in: 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005, pp. 635–638.

[43] M. dos Santos Soares, J. Vrancken, A. Verbraeck, User requirements modeling and analysis of software-intensive systems, Journal of Systems and Software 84 (2) (2011) 328–339.

[44] ISO/IEC 42010:2007 – originally IEEE Std 1471:2000 – Recommended Practice for Architectural Description of Software-intensive Systems (2007).

[45] P. Lago, P. Avgeriou, R. Hilliard, Guest editors' introduction: Software architecture: Framing stakeholders' concerns, IEEE Software 27 (2010) 20–24.

[46] P. Clements, R. Kazman, M. Klein, Evaluating Software Architecture: Methods and Case Studies, Boston: Addison-Wesley, 2002.

[47] T. Yue, L. Briand, Y. Labiche, A use case modeling approach to facilitate the transition towards analysis models: Concepts and empirical evaluation, in: 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09), 2009, pp. 484–498.

[48] A. Weissman, M. Petrov, S. Gupta, A computational framework for authoring and searching product design specifications, Advanced Engineering Informatics 25 (3) (2011) 516–534.

[49] L. Briand, D. Falessi, S. Nejati, M. Sabetzadeh, T. Yue, Traceability and sysml design slices to support safety inspections: A controlled experiment, Tech. Rep. 2010-08, Simula Research Laboratory (2010).

[50] S. Maoz, J. Ringert, B. Rumpe, ADDiff: semantic differencing for activity diagrams, in: FSE, 2011, pp. 179–189.

Appendix A

Theorem 1. The block diagram and activity diagram slices generated by the algorithm in Figure 10 satisfy the conditions in Definitions 4 and 5, respectively.

Proof:
Our proof consists of the following sub-arguments:

- $\forall b \in B_r \cdot \exists b' \in B_s \cdot (bOp \cap Block\_Op_r \subseteq bOp') \wedge (bAttr \cap Block\_St_r \subseteq bAttr')$ where bOp is the set of operations of b, bOp' is the set of operations of b', $bAttr$ is the set of attributes of b, and $bAttr'$ is the set of attributes of b'.

  $\forall b \in B_r \cdot \exists b' \in B_s \cdot \forall x \cdot x \in (bOp \cap Block\_Op_r)$
  $\Rightarrow$ (by properties of $\cap$)
  $\forall b \in B_r \cdot \exists b' \in B_s \cdot \forall x \cdot x \in bOp \wedge x \in Block\_Op_r$
  $\Rightarrow$ (by line 1 of algorithm in Figure 10 and definition of $Block\_Elem_r$)
  $\forall b \in B_r \cdot \exists b' \in B_s \cdot \forall x \cdot x \in bOp \wedge x \in Design\_Elem_r$
  $\Rightarrow$ (by line 10 of algorithm in Figure 10)
  $\forall b \in B_r \cdot \exists b' \in B_s \cdot \forall x \cdot x \in bOp'$

  $\forall b \in B_r \cdot \exists b' \in B_s \cdot \forall x \cdot x \in (bAttr \cap Block\_St_r)$
  $\Rightarrow$ (by properties of $\cap$)
  $\forall b \in B_r \cdot \exists b' \in B_s \cdot \forall x \cdot x \in bAttr \wedge x \in Block\_St_r$
  $\Rightarrow$ (by line 1 of algorithm in Figure 10 and definition of $Block\_Elem_r$)
  $\forall b \in B_r \cdot \exists b' \in B_s \cdot \forall x \cdot x \in bAttr \wedge x \in Design\_Elem_r$
  $\Rightarrow$ (by line 11 of algorithm in Figure 10)
  $\forall b \in B_r \cdot \exists b' \in B_s \cdot \forall x \cdot x \in bAttr'$

- $\forall ap \in AP' \cdot \exists ap' \in AP_s \cdot ap = (N, n_0, E) \wedge ap' = (N', n_0', E') \Rightarrow (N \cap Act\_Nd_r \subseteq N' \subseteq N) \wedge (E \cap Act\_Ed_r \subseteq E')$.

$x \in N \cap Act\_Nd_r$
$\Rightarrow$ (by properties of $\cap$)
$x \in N \wedge x \in Act\_Nd_r$
$\Rightarrow$ ($N = aNodes$ by line 16 of the algorithm in Figure 10)
$x \in aNodes \wedge x \in Act\_Nd_r$
$\Rightarrow$ (by line 1 of algorithm in Figure 10 and definition of Act_Elem_r)
$x \in aNodes \wedge x \in Design\_Elem_r$
$\Rightarrow$ (by line 18 of algorithm in Figure 10)
$x \in aNodes'$
$\Rightarrow$ ($N' = aNodes'$ by line 20 of the algorithm in Figure 10)
$x \in N'$

$x \in N'$
$\Rightarrow$ ($N' = aNodes'$ by line 20 of the algorithm in Figure 10)
$x \in aNodes'$
$\Rightarrow$ (by line 18 of algorithm in Figure 10)
$x \in aNodes$
$\Rightarrow$ ($N = aNodes$ by line 16 of the algorithm in Figure 10)
$x \in N$

$x \in E \cap Act\_Ed_r$
$\Rightarrow$ (by properties of $\cap$)
$x \in E \wedge x \in Act\_Ed_r$
$\Rightarrow$ ($E = aEdges$ by line 16 of the algorithm in Figure 10)
$x \in aEdges \wedge x \in Act\_Ed_r$
$\Rightarrow$ (by line 1 of algorithm in Figure 10 and definition of Act_Elem_r)
$x \in aEdges \wedge x \in Design\_Elem_r$
$\Rightarrow$ (by line 19 of algorithm in Figure 10)
$x \in aEdges'$
$\Rightarrow$ ($E' = aEdges'$ by line 20 of the algorithm in Figure 10)
$x \in E'$

- There is at least one path from n_0 to n'_0 (or to all successors of n'_0) that does not go through any node in N'. This directly follows from lines 24 to 28 of the algorithm in Figure 10, and the fact that $N' = aNodes'$ by line 20 of the algorithm in Figure 10.

□

Appendix B

Proof of Soundness of the Algorithm in Section 5.2

In this appendix, we argue that the slicing algorithm in Figure 10 generates activity diagram slices that are sound for verifying temporal safety requirements. To do so, we first define a temporal semantic for safety requirements and activity partitions. Let Σ be an alphabet. We define a *trace* σ over Σ to be a finite sequence $\sigma_0\sigma_1 \ldots \sigma_n$, where $\forall i \cdot 0 \leq i \leq n, \sigma_i \in \Sigma$. We denote by Σ^* the set of all finite traces over Σ.

Definition 6 (Temporal Safety Requirements). *Let Σ be an alphabet. We define a temporal safety requirement to be a trace $r \in \Sigma^*$.*

For example, the trace formalizing the requirement in Figure 8 is

$\texttt{go\_load\_position}() \cdot \texttt{feed\_table}() \mid \texttt{initialize}() \cdot \texttt{feed\_table}()$

where Σ is the set of block operations in Figure 6(a). In our work, the alphabet Σ underlying a requirement r corresponds to the set of block operations and block states related to r, i.e., $Block\_Elem_r$.

In Definition 3, we described the syntax of activity partitions. Below, we provide a more detailed definition for the syntax of activity partitions which augments the syntax already given in Definition 3 with labels for activity nodes.

Definition 7 (Activity Partition (Syntax)). *An activity partition ap is a tuple (Σ, N, n_0, E, L), where Σ is a set of activity node labels, N is a finite set of activity nodes, $n_0 \in N$ is an initial activity node, $E \subseteq N \times N$ is a set of activity edges, and $L : N \rightarrow \Sigma$ is a labelling function.*

We define the semantics of an activity diagram as the set of traces of activity node labels that can be generated from the initial node of that diagram. Note that SysML activity diagrams have two kinds of nodes: control nodes, (i.e., those describing actions/activities) and object nodes (i.e., those describing data). We first argue that in activity diagrams conforming to our methodology, object nodes can be replaced with control nodes. This is because according to Steps 6 and 7 of our methodology in Section 3, we use object nodes in activity diagrams to model signal communications only (i.e., sending and receiving of signals). Specifically, an object node for sending or receiving of a signal, `sig`, can be replaced by a

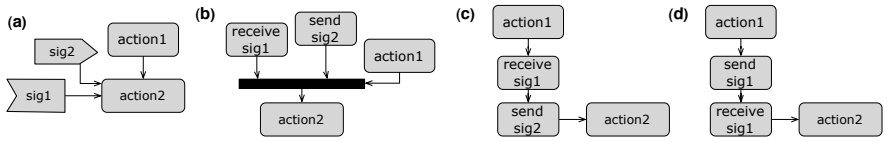

Figure 18: Transforming object nodes used for sending and receiving of signals to control nodes

control node labelled `send sig` or `receive sig`, respectively. Figure 18 shows an example of our translation. The two object nodes for receiving sig1 and sending sig2 in Figure 18(a) are replaced by two control nodes describing the actions for receiving sig1 and sending sig2 in Figure 18(b), respectively. According to the semantics of SysML activity diagrams, an action/activity node becomes active when it has received all its input tokens, i.e., signal tokens or control tokens. So a synchronization bar is used in Figure 18(b) to capture the fact that all the activity nodes (action1, send sig2, and receive sig1) must complete before action2 starts. The semantics of the example activity diagram in Figure 18(b) is then the possible inter-leavings of the parallel actions (action1, send sig2, and receive sig1) followed by action2. Two possible traces characterizing the semantics of the activity diagram in Figure 18(b) are shown in Figures 18(c) and (d). Our definition of activity diagram semantics (given below) is very similar to the operational (trace-based) semantics for activity diagrams given in [50]. The main difference is that their semantic definition includes variables, while in our work data elements are limited to signals, and hence, we do not formalize variables in our trace-based semantics.

Definition 8 (Activity Partition (Temporal Semantics)). *Let* $ap = (\Sigma, N, n_0, E, L)$. *A trace* $\sigma = \sigma_0\sigma_1...\sigma_k$ *is a behavior produced by* ap *iff there is a sequence* $n_0n_1...n_{k+1}$ *of activity nodes s.t. for every* $0 \leq j \leq k$, $(n_j, n_{j+1}) \in E$ *and* $L(n_j) = \sigma_j$. *The set of behaviors of* ap, $\mathcal{L}(ap)$, *is the set of all traces that can be produced by* ap.

Note that in the semantics of activity partitions, we treat the merge/split nodes as nodes with ϵ label, indicating a *silent* action. For example, the set of activity node labels for the `FeedBelt` activity partition in Figure 6(b) is $\{$ `receive turnON`, `receive AddBlank`, `receive FeedTable`, `initialize`, `add_blank`, `feed_table`, `send Go_Unload_Position`$\}$. The initial node of this activity partition is `receive turnON`. An example of a behavior of this activity diagram is

`receive turnON` · `initialize` · `add_blank` · `feed_table` · `send Go_Unload_Position`

In Definition 5, we provided a definition for activity partition slices. Here, we extend that definition with conditions on the activity node labels.

Definition 9 (Activity Partition Slices (Syntax)). *Let $ap = (\Sigma, N, n_0, E, L)$ be an activity partition. Let r be a safety requirement over the set of alphabet Σ_r. An activity partition slice of ap with respect to a safety requirement r is $(\Sigma', N', n_0', E', L')$ where N', n_0', and E' satisfy the conditions in Definition 5, and in addition, we have*

1. $(\Sigma_r \cap \Sigma) \subseteq \Sigma' \subseteq \Sigma$,
2. $L' \subseteq L$

For example, the activity partitions in Figure 11(b) are slices of the partitions in Figure 6(b) with respect to the requirement trace:

`receive turnON` · `initialize` · `add_blank` · `feed_table` · `send Go_Unload_Position`

The set of alphabet for the activity diagram slice for `FeedBelt` is {`receive FeedTable`, `feed_table`, `sendGo_Unload_Position`} which is a subset of the alphabet of the activity diagram for `FeedBelt` in Figure 6(b) and a superset of the alphabet of the requirement trace when it is constrained by the alphabet of `FeedBelt`. Similarly, the set of alphabet for the activity diagram slice for `Table` is {`send FeedTable`, `go_load_position`, `initialize`} which is a subset of the alphabet of the activity diagram for `Table` in Figure 6(b) and a superset of the alphabet of the requirement trace when it is constrained by the alphabet of `Table`.

An activity partition slice ap_r is temporally sound if its set of behaviors $\mathcal{L}(ap_r)$ is preserved in the set of behaviors of its corresponding original (non-sliced) activity partition $\mathcal{L}(ap)$. Hence, any temporal requirement that holds over the slice will hold over the original design as well. To prove this argument, we note that some of the activity node labels of the original activity partition are abstracted away in the slice. Hence, we first define a notion of projection on the temporal traces to formalize the act of abstracting away some labels from a trace.

Definition 10 (Trace Projection). *Let $\Sigma' \subseteq \Sigma$ be an alphabet, and $\sigma = \sigma_0 \ldots \sigma_n$ be a trace over Σ. The* projection *of σ to Σ', denoted $\sigma \downarrow_{\Sigma'}$, is defined as:*

$$\sigma \downarrow_{\Sigma'} = (\sigma_0 \downarrow_{\Sigma'})(\sigma_1 \downarrow_{\Sigma'})...(\sigma_n \downarrow_{\Sigma'})$$

where $\sigma_i \downarrow_{\Sigma'} = \sigma_i$ if $\sigma_i \in \Sigma'$, and ϵ otherwise. Further, let $\mathcal{T} \subseteq \Sigma^*$. The projection *of $\mathcal{T}$ to Σ' is denoted $\mathcal{T} \downarrow_{\Sigma'}$ and is defined as:*

$$\mathcal{T} \downarrow_{\Sigma'} = \{\sigma \mid \exists \sigma' \in \mathcal{T} \cdot \sigma = \sigma' \downarrow_{\Sigma'}\}$$

Theorem 2. *Let $ap = (\Sigma, N, n_0, E, L)$ be an activity partition, let r be a temporal safety requirement with the set of alphabet $\Sigma_r \subseteq \Sigma$, and let $ap_r = (\Sigma', N', n_0', E', L')$ be an activity partition slice of ap with respect to r generated by the algorithm in Figure 10. Then, ap_r is a sound activity diagram slice of ap. That is,*

I $(\Sigma_r \cap \Sigma) \subseteq \Sigma' \subseteq \Sigma$ *and* $L' \subseteq L$.

III $L(ap_r) \subseteq L(ap) \downarrow_{\Sigma'}$

Proof:
Our proof has the following sub-arguments:

- $(\Sigma_r \cap \Sigma) \subseteq \Sigma'$

  $e \in (\Sigma_r \cap \Sigma)$
  $\Rightarrow$ (By Definitions 1 and 9)
  $\exists n \cdot n \in aNodes \wedge L(n) = e \wedge n \in Design\_Elem$
  $\Rightarrow$ (by Line 18 of the algorithm, and
  that the algorithm does not change the activity node labels)
  $\exists n \cdot n \in aNodes' \wedge L'(n) = e$
  $\Rightarrow$ ($N' = aNodes'$ by line 20 of the algorithm)
  $\exists n \cdot n \in N' \wedge L'(n) = e$
  $\Rightarrow$ (By Definition 9)
  $e \in \Sigma'$

- $\Sigma' \subseteq \Sigma$

  $e \in (\Sigma')$
  $\Rightarrow$ (By Definition 9)
  $\exists n \cdot n \in N' \wedge L'(n) = e$
  $\Rightarrow$ (by $N' \subseteq N$ from Theorem 1, and
  that the algorithm does not change the activity node labels)
  $\exists n \cdot n \in N \wedge L(n) = e$
  $\Rightarrow$ (By Definition 9)
  $e \in \Sigma$

- $L' \subseteq L$. This follows from the fact that in our algorithm we never manipulate the node labels.
- $\mathcal{L}(ap_r) \subseteq \mathcal{L}(ap) \downarrow_{\Sigma'}$

$\sigma \in \mathcal{L}(ap_r)$
$\Rightarrow$ (By Definition 8)
$\exists \sigma_0, \ldots, \sigma_{k+1} \in \Sigma' \cdot \exists n_0 n_1 ... n_{k+1} \in N' \cdot \forall 0 \leq j \leq k \cdot (n_j, n_{j+1}) \in E' \wedge L'(n_j) = \sigma_j$
$\Rightarrow$ (By $N' \subseteq N$, $L' \subseteq L$, and $\Sigma' \subseteq \Sigma$)
$\exists \sigma_0, \ldots, \sigma_{k+1} \in \Sigma \cdot \exists n_0 n_1 ... n_{k+1} \in N \cdot \forall 0 \leq j \leq k \cdot (n_j, n_{j+1}) \in E' \wedge L(n_j) = \sigma_j$
$\Rightarrow$ (Case 1: $(n_j, n_{j+1}) \in E$)
$\exists \sigma_0, \ldots, \sigma_{k+1} \in \Sigma \cdot \exists n_0 n_1 ... n_{k+1} \in N \cdot \forall 0 \leq j \leq k \cdot (n_j, n_{j+1}) \in E \wedge L(n_j) = \sigma_j$
$\Rightarrow$ (By Definitions 8 and 10)
$\sigma \in \mathcal{L}(ap) \downarrow_{\Sigma'}$
$\Rightarrow$ (Case 2: (n_j, n_{j+1}) is a stuttering edge)
$\exists n_j, m_0, \ldots, m_t, n_{j+1} \in N \cdot \forall 0 \leq l < t \cdot (n_j, m_0) \in E \wedge (m_l, m_{l+1}) \in E \wedge$
$(m_t, n_{j+1}) \in E \wedge L(m_l), L(m_{l+1}) \in \Sigma \setminus \Sigma'$
$\Rightarrow$ (By Definitions 8 and 10)
$\sigma \in \mathcal{L}(ap) \downarrow_{\Sigma'}$

□