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

Shiva Nejati[†] Mehrdad Sabetzadeh[†] Davide Falessi[†] Lionel Briand[†] Thierry Coq[‡]

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

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

Abstract

Context: Traceability is one of the basic tenets of all software safety standards and a key prerequisite for certification of software. Despite this, the safety-critical software industry is still suffering from a chronic lack of guidelines on traceability. An acute traceability problem that we have identified through observing software safety certification processes has to do with the link between safety requirements and software design. In the current state of practice, this link often lacks sufficient detail to support the systematic inspections conducted by the certifiers of the software safety documentation. As a result, the suppliers often have to remedy the traceability gaps after the fact which can be very expensive and the outcome often is far from satisfactory.
Objective: The objective of this article is developing a framework to enable systematic and efficient software design inspections during safety certification. In particular, the framework enables safety engineers and certifiers to extract design slices (model fragments) that filter out irrelevant details but keep enough context information for the slices to be easy to inspect and understand. This helps reduce cognitive load and thus makes it less likely that serious safety issues would be overlooked.
Method: 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 safety requirement. The framework is implemented in a tool, named SafeSlice, that supports establishing the traceability links envisaged by the methodology, automated consistency checking of these links, and automated generation of SysML design slices.
Results: We provide a formal proof that our slicing algorithm is sound for temporal safety properties, and argue about the completeness of the slices based on our practical experience. We report on the lessons learned from applying our approach to two case studies, one benchmark case and one industrial case. 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.

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.

The work we report in this paper 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.

A particularly acute 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 stringent inspections that certifiers conduct to ensure that the (software-related) safety objectives of a system are adequately addressed by the software design. Consequently, the suppliers often have to recover the missing traceability information after the fact, which usually turns out to be a very costly endeavor with unpredictable and less than satisfactory outcomes.

1.1. Contributions

In this paper, we are going to address the problem of traceability from requirements to design in the context of safety certification. 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 be achieved from traceability. In our work, requirements are expressed as unrestricted natural language statements, design is expressed using Systems Modeling Language (SysML) [4], and the goal is to enable systematic design safety inspections. This combination is novel and increasingly common in practice, noting that SysML is an INCOSE standard and represents a significant and increasing segment of the industry. Specifically, we make the following contributions:

- We characterize, based on our observation of software safety certification inspections, the traceability information that is necessary for arguing that the safety requirements are indeed accounted for in the software design. This is achieved by developing an *information model* for traceability.

- We provide guidelines on how the traceability links prescribed by our information model can be established during SysML modeling.

- We define a formal mechanism through which design slices relevant to a safety requirement can be automatically extracted to minimize the effort required to check design compliance. This facilitates both impact analysis and the certification's usual workflow of activities.

- We develop tool support for our methodology. Our tool, named SafeSlice (`http://modelme.simula.no/pub/pub.html#ToolSlice`), enables establishing the traceability links envisaged by the methodology, automated consistency checking of these links, and automated generation of SysML design slices. Additionally, the tool provides a range of facilities for advanced model navigation, report generation, and project status monitoring.

- We describe the steps we have taken to validate our approach. In particular, we provide a formal proof that our slicing algorithm is sound for temporal 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 IO software module used on ships and offshore facilities. 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 reminder of the paper is structured as follows: In Section 2, we describe some root causes of certification problems. In Section 3, we provide a short overview of SysML. We propose, in Section 4, a modeling methodology aimed at facilitating safety certification inspections. The information model upon which the traceability links in the methodology are based is given in Section 5. In Section 6, 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 7 presents tool support. Section 8 provides an evaluation of our approach. Section 9 reviews the related work; and Section 10 summarizes the paper and outlines directions for future work.

2. Software Safety Certification

To better understand current software safety certification practices, we attended three certification meetings (totaling approximately 9 hours) between a certification body and a supplier of software-intensive safety-critical systems in the maritime and energy sector. The purpose of these meetings was to determine if certain software components of the system being certified were compliant to IEC 61508 up to SIL3[1]. The meetings we attended were exclusively focused on ensuring the quality of the requirements, design, and architecture specifications for the software components. Showing compliance to IEC 61508 involves satisfying a variety of other criteria having to do with V&V plans and results, the software process, configuration management, handling of faults, developers' competence, etc. These aspects are orthogonal to our work in this paper and were not the subject of the meetings we attended.

The communication protocol between the supplier and the certifier was as follows: The supplier would first send a (versioned) package composed of the requirements, design, and architecture specifications to the certifier. This was done long in advance of a scheduled certification meeting, so that the certifier would have sufficient time to conduct a thorough review of the specifications and provide feedback. The certifier's feedback came in the form of a detailed list of issues to be discussed during a meeting and was provided to the supplier well ahead of the meeting. During the meeting, these issues were looked at one by one and the supplier outlined its plan on how it was going to address each issue in subsequent versions of the specifications. The updated specifications would then again be reviewed by the certifier, and the process would continue iteratively until the certifier was fully satisfied with the specifications.

A total of 66 distinct issues were discussed in the meetings we attended. Of these, 21 (32%) were minor and did not entail any changes to the substance of the specifications. Examples included terminology discrepancies, and minor omissions and misinterpretations. Five issues (7%) had to do with safety requirements that were not explicitly stated in the specifications or requirements that were introduced recently into the regulations and had not yet been accounted for by the supplier. Identifying this class of issues needed significant domain expertise from the certifier's side and familiarity with the range of applicable standards. The remaining 40 issues (61%) arose due to a combination of the following factors:

- **Traceability.** The certifier demanded traceability (1) between requirements and their sources (e.g. standards, stakeholders); (2) between system-level and software-level requirements; (3) between requirements and environ-

[1]IEC 61508 specifies 4 levels of safety. These are called Safety Integrity Levels (SILs). SIL1 is the lowest and SIL4 is the highest level.

mental assumptions (4) between software requirements and software blocks; and (5) between software blocks at different levels abstraction (e.g., subsystem, module, component). These links were either missing, or hard to find in the specifications due to inadequate structuring (described below).

- **Structuring of the specifications.** Requirements decomposition was not explicit; the software workflows were difficult to understand and at times ambiguous; interaction interfaces between system components and between the system and the users were not clearly delineated; and getting an overall view of the software architecture from the descriptions required significant effort. The main cause of these problems was the specifications being mostly text-based.

This last category of certification issues – concerning traceability and the structuring of the specifications – can largely be avoided by developing an appropriate development methodology based on models. Doing so is the focus of the remainder of this paper. As we stated earlier, addressing traceability without requiring more structure and precision on the development artifacts is unlikely to succeed because the traceability links cannot be defined at an adequate level of rigor over poorly structured and imprecise artifacts.

A final note to add here is that the percentages given in this section might not be representative because we only considered one certification project. In fact, we anticipate variations if the study was repeated with different parameters, e.g., a different safety integrity level, different safety standard, different system, or different certification body. Further, we did not measure the costs incurred over addressing each class of issues, although we observed that all the non-minor issues raised by the certifier were very expensive to fix. While additional studies of certification processes are needed, we believe the current study lends support to the anecdotal evidence that traceability is a major “bottleneck” during software safety certification.

3. 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 [5], 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 [6]:

- SysML expresses systems engineering semantics (interpretations of modeling constructs) better than UML, thereby reducing the bias UML has towards software. In particular, UML classes are replaced with a concept called *block* in SysML. Block is a *modular unit of system description*. Blocks are used to describe structural concepts in a system and its environment.

- SysML has built-in *cross-cutting* links for interrelating requirement and design elements. This allows engineers to relate requirements and design elements/models described at different levels of abstraction.

Our methodology in Section 4 utilizes all SysML diagrams. For a complete specification of SysML, see [7].

4. Design Methodology

SysML is only a modeling notation and does not provide a methodology on how to model a system (or system-to-be). To be able to effectively apply SysML, one needs a specific methodology tailored to the problem domain and the objectives to be achieved from modeling. In this section, we describe a methodology for modeling safety critical control systems. Our methodology is based on SysML and adapts the best practices in the existing model-based systems engineering methods [8] to the needs of software safety certification. The main objective pursued from modeling is to facilitate safety inspections by providing precise and

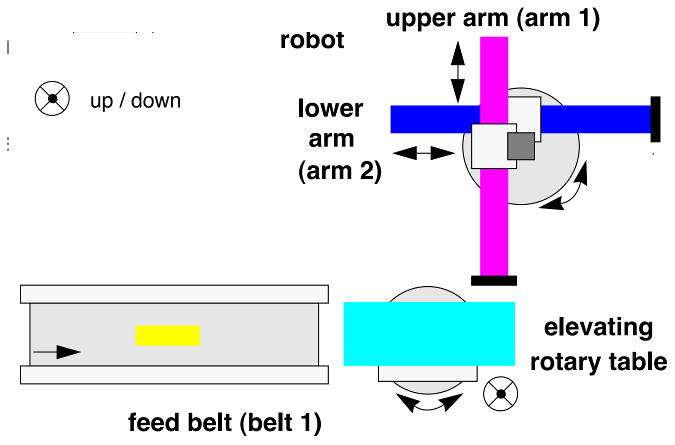


Figure 1: A fragment of the Production Cell System (PCS) [9]

unambiguous specifications of the system requirements and design (structure and behavior) and establishing traceability links between requirements and design. These links provide the evidence one needs for arguing that the requirements are properly addressed by the design.

We use a small fragment of a Production Cell System (PCS) [9] 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 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

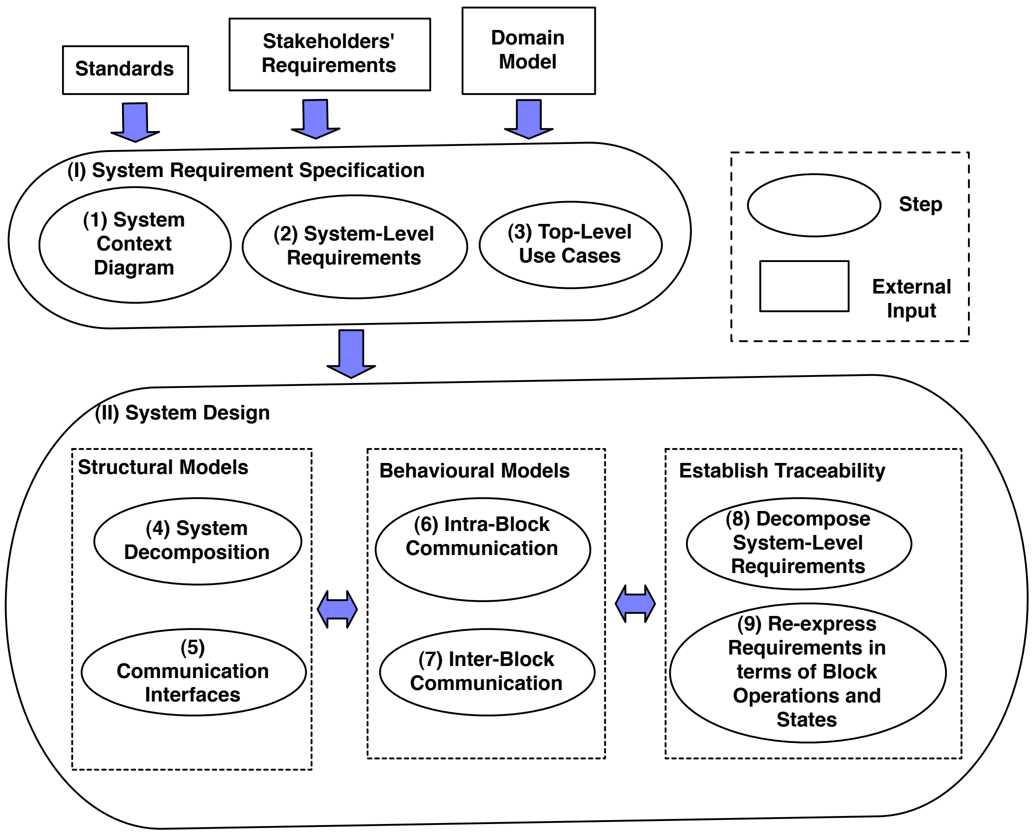

Standards
Stakeholders' Requirements
Domain Model
(I) System Requirement Specification
(1) System Context Diagram
(2) System-Level Requirements
(3) Top-Level Use Cases
Step
External Input
(II) System Design
Structural Models
(4) System Decomposition
(5) Communication Interfaces
Behavioural Models
(6) Intra-Block Communication
(7) Inter-Block Communication
Establish Traceability
(8) Decompose System-Level Requirements
(9) Re-express Requirements in terms of Block Operations and States

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

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, but 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.

There are two main characteristics that distinguish our methodology from other MDE-based methodologies for systems engineering:

1. The decomposition of system-level requirements is interleaved with the de-

sign 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 specific guidelines for capturing requirement and design details 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 [10].

Phase I - Requirements Specification. As shown in Figure 2, this phase has three steps described below:

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, either through the hardware devices or through the software embedded in the hardware devices. We use SysML blocks to represent the context.

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 [11]. Therefore, it is important that the descriptions of these assumptions allow systematic analysis. This support the assessment of the validity of requirements, specifications, and design decisions and to verify that there are no conflicts between the required system properties. We describe the environmental assumptions using SysML parametric diagrams (for continuous properties of a hardware entity), or OCL constraints (for discrete properties of a software entity). We further create traceability links from assumptions to system-level requirements.

In Figure 3, 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 feed belt that conveys the blank to the press, and that the width of a blank is not larger than 3/4 of the width of the feed belt. The former

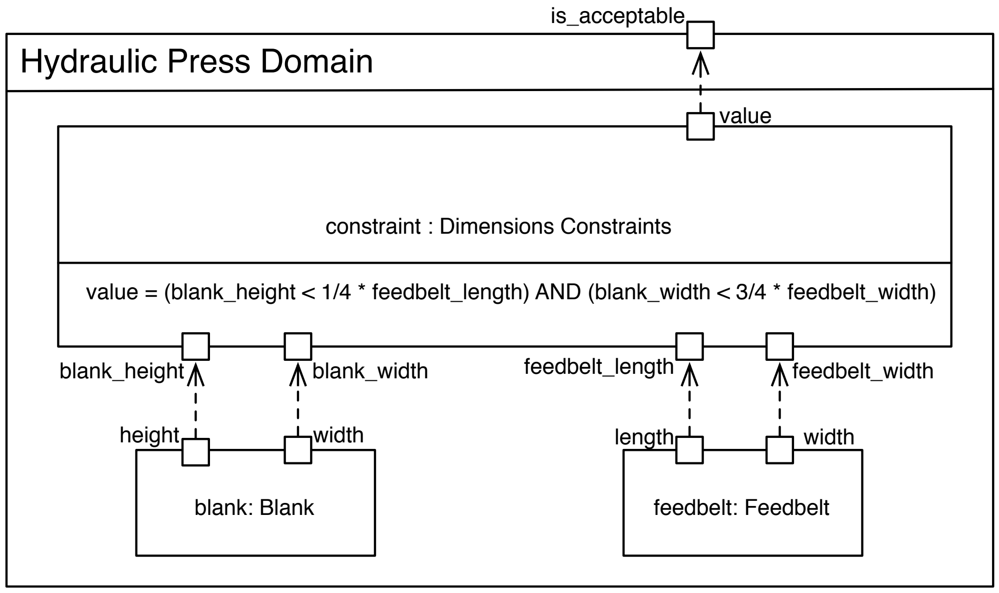


Figure 3: A SysML parametric diagram expressing an assumption.

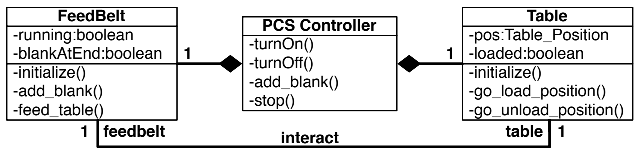
constraint is to ensure that the blank is small enough to be picked up by the robot arm that places the blank on the press, and the latter – to ensure that blanks would not fall off the edges of the feed belt while in motion.

Step 2: System-level requirement diagram. System-level requirements address the entire system, hardware or software. These requirements may come from a variety of sources including, standards, customers, 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 [12] (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.

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. They can be directly traced to the system-level requirements that they are expected to address.

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 its constituent blocks and their interaction points, and behavioral views describe how the blocks work together and communicate with

(a)



(b)

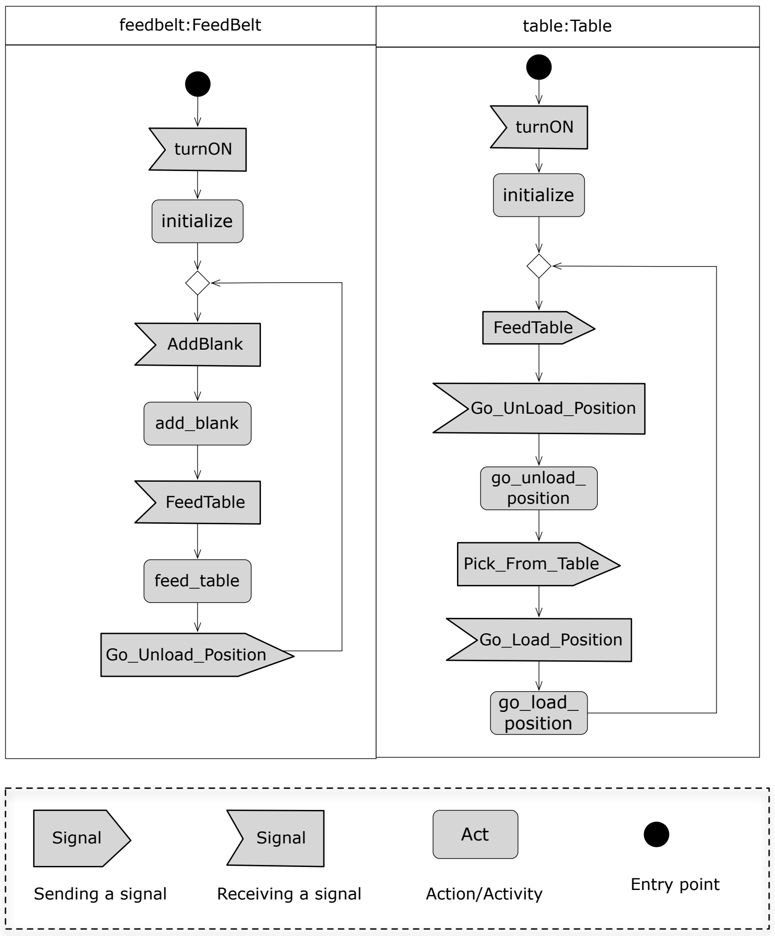


Figure 4: A fragment of design diagrams for PCS: (a) Block Definition Diagram, and (b) Activity partitions for the blocks in (a).

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). 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 4(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. 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 BDD into a set of architectural connectors with specific communication interfaces and ports. Making the interfaces between different system blocks explicit is a major concern in safety-critical systems to ensure that components can be integrated properly. See the "insufficient structure and precision" issues in Section 2.

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. For each software block with control behavior, we create one state machine diagram. Furthermore, each such block is related to a timeline or 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 activity partitions and sequence diagram timelines. To keep the blocks consistent with the behavioral diagrams, the messages communicated between blocks in sequence diagrams, the actions in activity diagrams, and the transitions' triggers/effects in state machine diagrams must be added as operations to the appropriate blocks. For example, Figure 4(b) shows the activity partitions related to `FeedBelt` and `Table` blocks in Figure 4(a). The actions `initialize`, `add_blank`, and `feed_table` in the activity partition related to `FeedBelt` appear as `FeedBelt` operations, and similarly the actions `initialize`, `go_load_position`, and `go_unload_position` in the activity partition of `Table` appear as `Table` operations in Figure 4(a).

In the rest of the paper, to be consistent with our industrial case study in Section 8, we focus on activity diagrams for representing block behaviors. Figure 4(b) includes a small legend summarizing the notational elements used in our activity diagrams. Note that our treatment for activity diagrams can be easily generalized to other behavioral diagrams (see [10]).

Using behavioural 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 extract the following dependencies. We can identify that (1) A signal triggers an activity/action. For example, the `FeedTable` signal triggers the `feed_table` activity in Figure 4(b). (2) An activity/action triggers sending of a signal. For example, the `go_load_position` activity triggers sending of the `FeedTable` signal. (3) A signal sent from one activity partition is received by another partition. For example, the `FeedTable` signal sent from the activity partition related to `Table` is received by the activity partition related to `FeedBelt`. (4) An activity/action directly triggers another activity/action via an operation call relation. 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 4(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 triggers the `feed_table` activity.

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. 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 5 shows how a system-level requirement of the PCS example is decomposed into a block-level requirement which is traced to `FeedBelt` and `Table` blocks from Figure 4(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

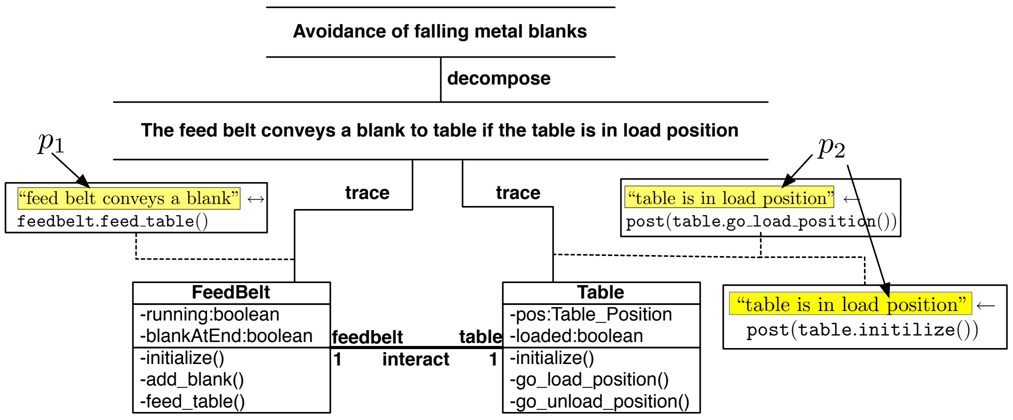

Avoidance of falling metal blanks
decompose
The feed belt conveys a blank to table if the table is in load position
p_1
p_2
trace
trace
"feed belt conveys a blank" ↔ feedbelt.feed_table()
"table is in load position" ← post(table.go_load_position())
"table is in load position" ← post(table.initilize())
FeedBelt
-running:boolean
-blankAtEnd:boolean
-initialize()
-add_blank()
-feed_table()
Table
-pos:Table_Position
-loaded:boolean
-initialize()
-go_load_position()
-go_unload_position()
feedbelt
table
1
interact
1

Figure 5: Traceability from a system-level requirement to a block-level requirement and from the block-level requirement to the relevant blocks.

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:

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

where, *mapping* and *rel* are non-terminals, and the rest, which are terminals, are explained below. We use the requirements and blocks in Figure 5 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: $(\texttt{block.attr}_1 = \texttt{v}_1 \wedge \ldots \wedge \texttt{block.attr}_n = \texttt{v}_n)$ e.g., $\texttt{table.pos} = \texttt{loadposition}$; 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 5 with mappings. The requirement in Figure 5 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 $\texttt{table.pos} = \texttt{loadposition}$, (2) via the post condition $\texttt{post(table.go\_load\_position())}$ as the operation $\texttt{go\_load\_position()}$ causes the table to move to its load position, (3) via the pre condition $\texttt{pre(table.go\_unload\_position())}$ as the operation $\texttt{go\_unload\_position()}$ assumes that the table is already in its load position, and (4) via the post condition $\texttt{post(table.initialize())}$ as the operation $\texttt{initialize()}$ 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) $p_1 \leftrightarrow \texttt{feedbelt.feed\_table()}$
(2) $p_2 \leftrightarrow \texttt{table.pos} = \texttt{loadposition}$
(3) $p_2 \leftarrow \texttt{post(table.go\_load\_position())}$
(4) $p_2 \leftarrow \texttt{pre(table.go\_unload\_position())}$
(5) $p_2 \leftarrow \texttt{post(table.initialize())}$

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 $\texttt{post(table.go\_load\_position())}$, $\texttt{pre(table.go\_unload\_position())}$, and $\texttt{post(table.initialize())}$, respectively.

In Figure 5, 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 safety requirements are already linked to the detection method, control, and actions specified for each failure mode during Failure Mode and Effects Analysis (FMEA) [13], and focus exclusively on requirements-to-design traceability, which was the main problem based on our analysis of actual certification meetings.

5. Traceability Information Model

The information model in Figure 6 specifies the well-formedness criteria for the traceability links underlying our methodology in Section 4. 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 6. (2) The traceability links that engineers manually create between entities: These are characterized by all the labelled associations in Figure 6. 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 4: These are characterized by the thick dashed-line associations in Figure 6. 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 4, 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 4, 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 4, Step 4).

- The `refine` link relating System-Level Safety Requirement and the `Use Case` operationalizing that requirement (see Section 4, Step 3).

- The `decompose` link relating System-Level Safety Requirement and Block-Level Safety Requirement (see Section 4, 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 4, Step 2). We also use the `derive` link 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 6.

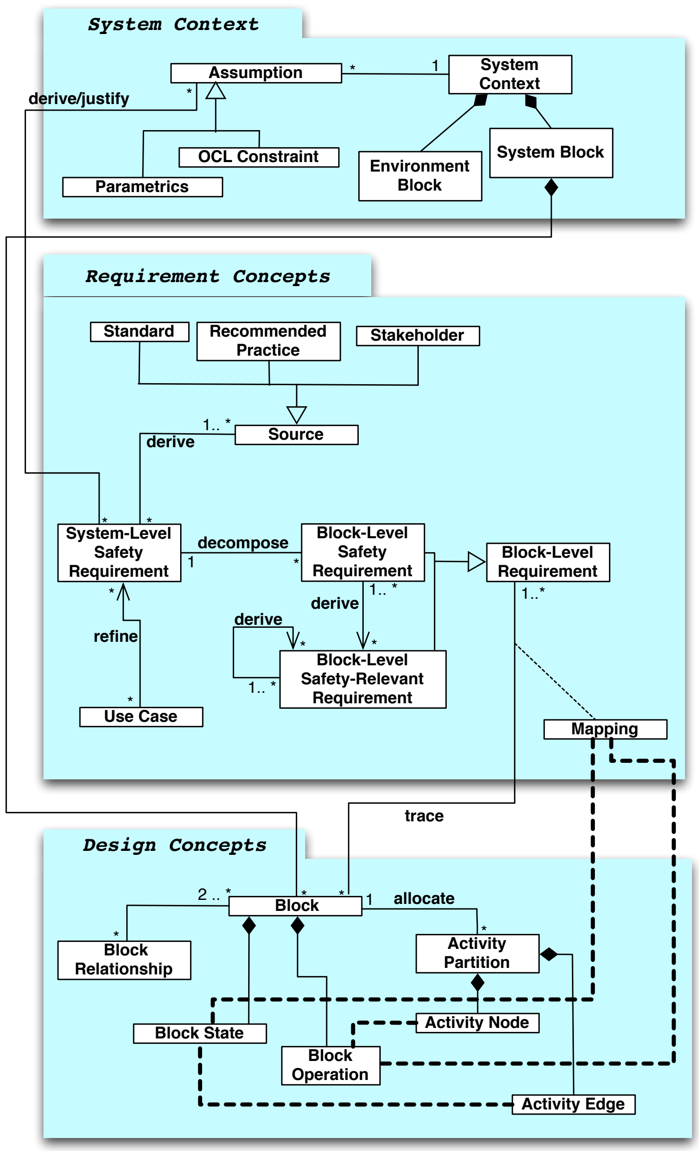


Figure 6: Traceability Information Model.

- The `trace` link between Block-Level Requirement and `Block` to indicate the blocks contributing to the satisfaction of those requirements (see Section 4, 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 6.

- The `allocate` link between `Block` and design elements representing the block behavior (see Section 4, Steps 6,7). To save space, in Figure 6, 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 [7]. The `decompose` link is used to relate 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.

Implied traceability links:

- As shown in Figure 6 and the example in Figure 5, 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 6). For example, in Figure 5, 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 causes the block to change its state, and hence appears in the diagrams representing its behavior. For example, the operation `go_load_position()` of `Table` in Figure 4(a) appears in the activity partition of `Table` in Figure 4(b). Note that block operations unrelated to safety requirements, e.g., getter operations of a block that simply retrieve values of some variables, do not necessarily appear as activity nodes. (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 4(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 6.

6. Automated Generation of Design Slices Relevant to Safety Requirements

This section explains how the traceability links described in Section 5 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 5 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. We provide a proof 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.

6.1. Slicing Algorithm

Figure 7 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 to evaluate the considered requirement. The algorithm takes as input a block-level safety or safety-relevant requirement r and a set of

SysML design diagrams conforming to the information model in Figure 6. It generates a block diagram slice, denoted $Block\_Slice_r$, as well as an activity diagram slice, denoted Act_Slice_r. Specifically, $Block\_Slice_r$ is a set of blocks and relations between blocks, and Act_Slice_r is a set of activity partitions. These sets are constructed such that all the blocks, the block operations/states, and the relations between blocks in $Block\_Slice_r$, and all the activity nodes/edges in Act_Slice_r directly or indirectly contribute to the satisfaction of r. The algorithm consists of an initialization phase and three main steps discussed and exemplified below.

Initialization. The algorithm starts by computing the sets B_r, Act_r, $Block\_Elem_r$, and Act_Elem_r from a set of SysML design diagrams that conform to the information model in Figure 6. For example, let r be the block-level requirement of Figure 5. For this requirement, B_r is $\{\texttt{Table}, \texttt{FeedBelt}\}$, Act_r contains the two activity partitions in Figure 4(b), $Block\_Elem_r$ is

$\{\texttt{feedbelt.feed\_table()}, \texttt{post(table.go\_load\_position())}, \texttt{post(table.initialize())}\}$

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 4(b). The algorithm also initializes Act_Slice_r and $Block\_Slice_r$ by setting them to $\emptyset$. 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())`.

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 6). 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 4), we discussed how the temporal relationships between block operations and activity nodes can be identified to compute $Design\_Elem_r$ (see

```
Algorithm. GENERATESLICE
Input:  A block-level safety requirement, $r$.
        A set of SysML design diagrams conforming to the information model in Figure 6.
Output: A block diagram slice related to $r$, $Block\_Slice_r$.
        An activity diagram slice related to $r$, $Act\_Slice_r$.
/* Initialization. */
    /*The trace and allocate links, the Mapping elements, and the relations between
    activity nodes and edges and block operations and states are represented in Figure 6.*/
1.  Let $B_r$ be the set of blocks related to $r$ via trace links.
2.  Let $Act_r$ be the set of activity partitions related to the blocks in $B_r$ via allocate links.
3.  Let $Block\_Elem_r$ be the set of block states and operations related to $r$ via Mapping elements.
4.  Let $Act\_Elem_r$ be the set of activity nodes and edges related to the elements in $Block\_Elem_r$.
5.  Let $Block\_Slice_r$ and $Act\_Slice_r$ be $\emptyset$

/* Step 1. Find elements temporally related to $r$ (Design_Elem$_r$). */
6.  $Design\_Elem_r = Block\_Elem_r \cup Act\_Elem_r$
7.  for any block $b \in B_r$ and any element $e \in Design\_Elem_r$ do
8.    if operation $op$ of $b$ triggers (or is triggered by) $e$ then
9.      $Design\_Elem_r = Design\_Elem_r \cup \{op\}$
10. for any activity partition $a \in Act_r$ and any element $e \in Design\_Elem_r$ do
11.   if activity node $n$ of $a$ triggers (or is triggered by) $e$ then
12.     $Design\_Elem_r = Design\_Elem_r \cup \{n\}$

/* Step 2. Extract block diagram slices (Block_Slice$_r$). */
13. for every block $b \in B_r$ do
14.   Let $bOp$ and $bAttr$ be the sets of operations and attributes of $b$, respectively.
      /*Remove any operation in $bOp$ that is not in $Design\_Elem_r$.*/
15.   $bOp' = bOp \cap Design\_Elem_r$
      /*Remove any attribute in $bAttr$ that is not in $Design\_Elem_r$.*/
16.   $bAttr' = bAttr \cap Design\_Elem_r$
17.   Let $bOp'$ and $bAttr'$ be the new sets of operations and attributes of $b$, respectively.
18.   $Block\_Slice_r = Block\_Slice_r \cup \{b\}$
    /* Add block relationships to the block diagram slice.*/
19. $Block\_Slice_r = Block\_Slice_r \cup \{rel \mid rel$ is a block relation between blocks in $B_r\}$

/* Step 3. Extract activity diagram slices (Act_Slice$_r$). */
20. for every activity partition $a \in Act_r$ do
21.   Let $aNodes$ and $aEdges$ be the sets of nodes and edges of $a$, respectively.
22.   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 $Design\_Elem_r$.*/
23.   $aNodes' = aNodes \cap \big(Design\_Elem_r \cup \{$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 $aNodes'$.*/
24.   $aEdges' = \{e \in aEdges \mid$ such that both ending points of $e$ are in $aNodes'\}$
25.   Let $aNodes'$ and $aEdges'$ be the new sets of nodes and edges of $a$, respectively.
      /* Add stuttering edges.*/
26.   for every pair $n, n' \in aNodes'$ do
27.     if $n'$ is reachable from $n$ in $a$ through edges none of which are in $aEdges'$ then
28.       add a stuttering edge from $n$ to $n'$
    /* Pick a new initial node.*/
29.   for every node $n$ in activity partition $a$ do
30.     if there is a path from $init$ to $n$ that does not go through any node in $aNodes'$ then
31.       mark $n$ as a new initial node of $a$.
32.   $Act\_Slice_r = Act\_Slice_r \cup \{a\}$
```

Figure 7: Algorithm for generating design slices.

[10] for more details). If the behavioural diagrams do not fully comply with our methodology, we can still build $Design\_Elem_r$ using existing techniques for control dependence analysis [14].

For example, $Design\_Elem_r$ for the requirement in Figure 5 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 4(b)
- the activity node related to sending signal `Go_Unload_Position` in the activity partition related to `FeedBelt` in Figure 4(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 Block_Slice$_r$). This step abstracts away attributes and operations not present in $Design\_Elem_r$ from blocks in B_r. It further removes the block relationships that are not between the blocks in B_r. For example, the block diagram slice related to the requirement in Figure 5 is shown in Figure 8(a).

Step 3 (Extract Act_Slice$_r$). This step abstracts away every activity node from activity partitions in Act_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* [15]. For example, the activity slice related to the requirement in Figure 5 is shown in Figure 8(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.

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 6.2). 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.

6.2. 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 certifiers can always rely on checking the generated design slices and never need to refer to the original

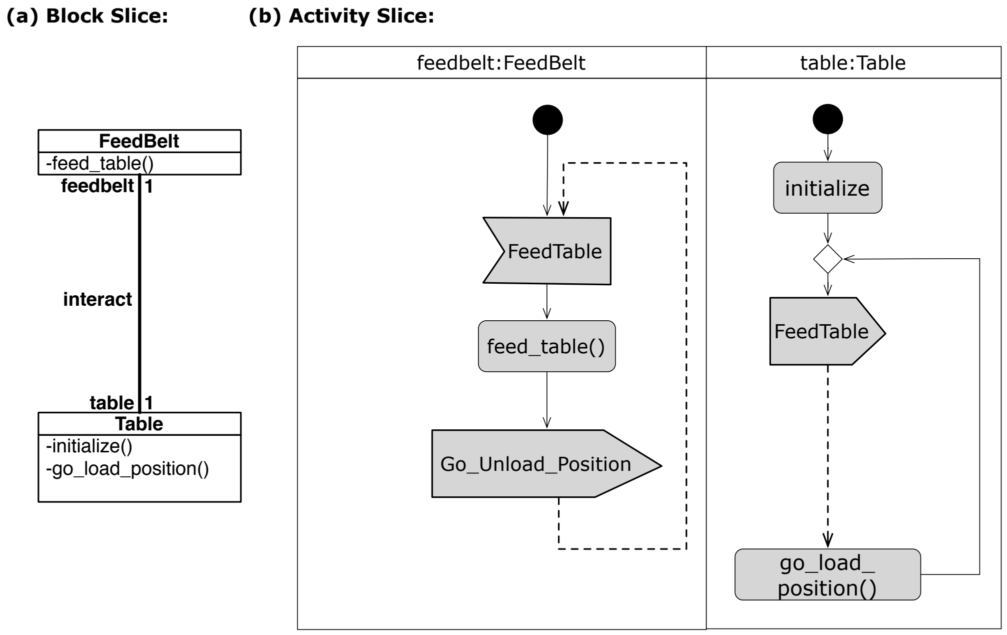


Figure 8: The block and activity slices for the requirement in Figure 5 extracted from the SysML design diagrams in Figure 4.

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 certifiers 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 [16]. For example, the requirement in Figure 5 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()`. Since 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 8(b) are the same as the orderings between these nodes in Figure 4(b), the slice in Figure 8(b) is sound for analysing the requirement in Figure 5. Below, we formally argue that the design activity slices generated by the algorithm in Figure 7 are sound for temporal constraints.

Temporal sequences. Let Σ be an alphabet. We define a *trace* σ over Σ to be a finite sequence $\sigma_0\sigma_1 \dots \sigma_n$, 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. Temporal safety requirements can be formalized as traces of block operations. For example, the trace formalizing the requirement in Figure 5 is

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

Activity partitions. An activity partition AP is a tuple (Σ, S, S_0, R, L), where Σ is a set of activity node labels, S is a finite set of activity nodes, $S_0 \subseteq S$ is a set of initial activity nodes, $R \subseteq S \times S$ is a transition relation, and $L : S \rightarrow \Sigma$ is a labelling function. A trace $\sigma = \sigma_0\sigma_1...\sigma_k$ is a behaviour produced by AP iff there is a sequence $s_0s_1...s_{k+1}$ of activity nodes s.t. $s_0 \in S_0$, and for every $0 \leq j \leq k$, $(s_j, s_{j+1}) \in R$ and $L(s_j) = \sigma_j$. The set of behaviours of AP, $L(AP)$, is the set of all traces that can be produced by AP. Note that in our formalization of activity partitions, we treat the nodes for sending and receiving of a signal, `sig`, as activities labelled `send sig` and `receive sig`, respectively. For example, the set of activity node labels for the `FeedBelt` activity partition in Figure 4(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 behaviour of this activity diagram is

$$\texttt{receive turnON} \cdot \texttt{initialize} \cdot \texttt{add\_blank} \cdot \texttt{feed\_table} \cdot \texttt{send Go\_Unload\_Position}$$

Activity partition slices. Let $AP = (\Sigma, S, S_0, R, L)$ be an activity partition. Let r be a safety requirement over the set of alphabet Σ_1. An activity partition slice of AP with respect to a safety requirement r is denoted by $AP_r = (\Sigma', S', S'_0, R', L')$ where $(\Sigma_1 \cap \Sigma) \subseteq \Sigma' \subseteq \Sigma$, $S'_0 \subseteq S' \subseteq S$, and $L' \subseteq L$. Further, the set of nodes in S' to which there is a path from a node in S_0 that does not go through any node in S' is S'_0. For example, the activity partitions in Figure 8(b) are slices of the partitions in Figure 4(b) with respect to the requirement trace:

`receive turnON` $\cdot$ `initialize` $\cdot$ `add_blank` $\cdot$ `feed_table` $\cdot$ `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 4(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 4(b) and a superset of the alphabet of the requirement trace when it is constrained by the alphabet of `Table`.

An activity diagram slice AP_r is temporally sound if its set of behaviours $L(AP_r)$ is preserved in the set of behaviours of its corresponding original (non-sliced) activity diagram $L(AP)$. 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. 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 1. *Let $AP = (\Sigma, S, S_0, R, L)$ be an activity diagram, let r be a temporal safety requirement with the set of alphabet $\Sigma_1 \subseteq \Sigma$, and let $AP_r = (\Sigma', S', S'_0, R', L')$ be an activity diagram slice of AP with respect to r generated by the algorithm in Figure 7. Then, AP_r is a sound activity diagram slice of AP. That is,*

I $(\Sigma_1 \cap \Sigma) \subseteq \Sigma' \subseteq \Sigma$*,* $S' \subseteq S$*, and* $L' \subseteq L$*.*

II *Any node in S' to which there is a path from S_0 that does not go through any node in S' is in S'_0.*

III $L(AP_r) \subseteq L(AP) \downarrow_{\Sigma'}$

Proof:
The argument **I** follows from the construction of slices in the algorithm in Figure 7. Specifically, the set of states and state labels of an activity diagram slice AP_r consists of the elements present in $Design\_Elem_r$, and further, $Design\_Elem_r$ is a subset of the set of states and state labels of the activity diagram AP. Moreover, $Design\_Elem_r$ contains all the elements that are explicitly present in the requirement r, i.e., Σ_1.

The argument **II** holds by lines 29–31 of the algorithm in Figure 7.

For the argument **III**, note that based on the construction in the algorithm in Figure 7, we have (1) the ending points of every (non-stuttering) transition in an activity slice are the same as the ending points of that transition in the original activity diagram; and (2) stuttering transitions replace sequences of consecutive transitions. By (1) and (2), it can be shown that the temporal ordering of transition labels are preserved, and hence, the set of traces of an activity diagram slice is a subset of the set of traces of its original corresponding activity diagram projected to the alphabet of the slice. □

The above theorem shows that our slicing algorithm in Figure 7 generates activity diagram slices that are sound for verifying temporal safety requirements. The semantics of activity diagrams is typically described using Petri nets [17], where an action becomes active when it has received all its input tokens, i.e., signal tokens or control tokens received upon completion of an activity. Here, we take a trace-based semantic approach by abstracting away the data that is being communicated, and interpreting sending and receiving of signals as actions (See formalization of Activity Partitions above). This is because a trace-based semantics allows us to reason about soundness with respect to temporal logic in a more convenient way than the token-flow semantics of Petri nets.

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 analysing 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 5 either of the mappings related to `post(table.go_load_position())` or `post(table.initialilize())`, the resulting activity partition slices in Figure 8(b) will not include the activity nodes `go_load_position` and `initialize` respectively. Second, the ability of certifiers to analyse 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 amount of information to verify certain requirements. Due to the subjectiveness of this issue, we plan to evaluate completeness 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 8 when sufficient traceability links and sufficient mapping elements are provided.

For example, we can argue that the block and activity diagram slices in Figure 8 contains enough information to check the requirement (r) in Figure 5. 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 8(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 5 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 8(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 necessary 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 8 are complete for analysing the requirement in Figure 5.

7. Tool Support

This section presents a tool, named SafeSlice (`http://modelme.simula.no/pub/pub.html#ToolSlice`), implementing our approach. Specifically, SafeSlice enables users to: 1) specify the traceability links envisaged by the traceability information model described in Section 5, 2) check the consistency of the established links, and 3) automatically extract slices of design with respect to requirements using the slicing algorithm in Section 6. These slices are in turn used for conducting inspections and ensuring that the design satisfies the safety requirements. In addition to implementing our proposed traceability mechanism and slicing algorithm, the tool provides facilities for managing inspections and report generation.

To make sure that our tool can be seamlessly applied in real development settings and easily maintained and improved, we built the tool as a plugin for an existing model-based development environment. Among the possible alternatives, we chose Enterprise Architect (EA) (`http://www.sparxsystems.com.au`) as the base UML/SysML modeling environment due to EA's usability, wide industrial adoption (confirmed by our industrial partners), availability of detailed guidelines for plugin construction, and built-in support for storing and linking heterogeneous development artifacts (natural language requirements specifications, UML/SysML models, Word documents, source code, etc.).

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 entirely in Visual C# and is roughly 10,000 lines of code excluding comments and third-party libraries.

Figure 9 shows the overall architecture of SafeSlice. It communicates asynchronously with EA via events. All the information related to a development project is stored by EA in a database. The plugin can read from and write to this database via EA's API. In particular, the additional traceability information required in our methodology, previously-generated design slices and reports, and the decisions made by users during inspections are all stored and retrieved by the plugin via the API; this communication layer assures reliability and it hides the underlying database technology. In the remainder of this section, we present the main features of SafeSlice: Rule Assistant, Slice Generator, and Inspection Assistant.

7.1. Rule Assistant

Rules Assistant is a feature of SafeSlice to help users with the correct application of the traceability information model presented in Section 5. Compliance with this information model is necessary for automatic generating of design slices. Rule Assistant provides support for checking the multiplicity and navigation constraints specified in the information model. These constraints are encoded using a spreadsheet to be easily modifiable by the user.

If a violation is detected, Rule Assistant provides diagnostic information about the violated rule, the model element(s) involved, and the action(s) to be taken to resolve the issue. For instance, in Figure 10, we show the diagnostic information generated by the Rule Assistant for the situation where a safety requirement has

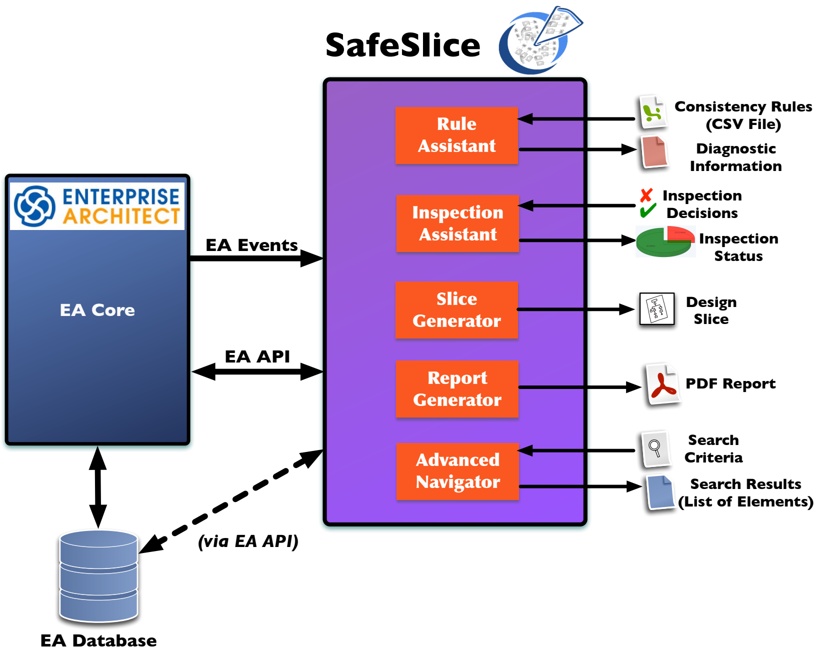
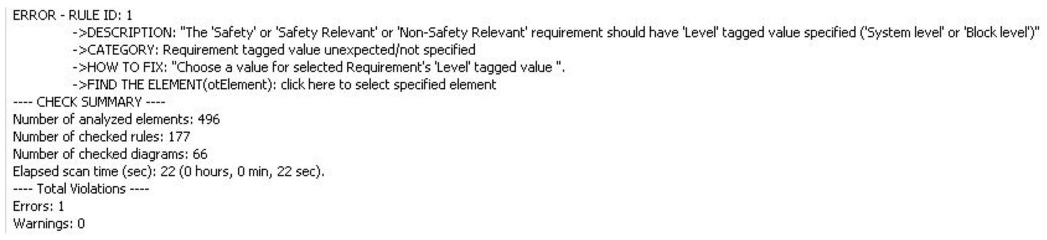


Figure 9: Architecture of SafeSlice



```
ERROR - RULE ID: 1
            ->DESCRIPTION: "The 'Safety' or 'Safety Relevant' or 'Non-Safety Relevant' requirement should have 'Level' tagged value specified ('System level' or 'Block level')"
            ->CATEGORY: Requirement tagged value unexpected/not specified
            ->HOW TO FIX: "Choose a value for selected Requirement's 'Level' tagged value ".
            ->FIND THE ELEMENT(otElement): click here to select specified element
---- CHECK SUMMARY ----
Number of analyzed elements: 496
Number of checked rules: 177
Number of checked diagrams: 66
Elapsed scan time (sec): 22 (0 hours, 0 min, 22 sec).
---- Total Violations ----
Errors: 1
Warnings: 0
```

Figure 10: Example Output from Rules Assistant.

been defined but it is unspecified whether the requirement is at the system level or at the block level. In this example, the user can read the "How to fix" field and then click on "Find the element" to quickly navigate to the element in question and apply the necessary fix.

Rule Assistant can check the compliance rules in two modes: investigator and listening. In the investigator mode, the compliance of an entire project is checked. In this mode, the user waits until the compliance checking process is finished and then applies the suggested changes if violations are detected. In the listening mode, Rule Assistant is always active in the background and monitors every change made by the user and checks that the change is consistent with the information model. If a violation is detected, feedback is immediately shown in the the diagnostic window. The time required by Rule Assistant to check all the rules is small. On a standard laptop, it took less than half a minute to check a SysML design with about a thousand elements (blocks, relations, activities, transitions, states, attributes, and operations) against all the rules.

7.2. Slice Generator

Slice Generator is an implementation of the slicing algorithm in Section 6. As we discussed in Section 5, a system safety requirement is refined into safety requirements at the level of blocks. If necessary, more detailed safety-relevant requirements could be defined in order to satisfy block-level safety requirements. In our tool, slices are constructed for atomic requirements, i.e., requirements that are directly related to design via traceability links. System-level requirements are never related to the design directly, because even in the simplest case where these requirements do not need to be decomposed, they still need to be allocated to some block and restated as a block-level requirement. If a block-level safety requirement is atomic, then our tool will generate a slice directly for the block-level requirement; otherwise, one slice will be generated for each of the atomic safety-relevant requirements contributing to the satisfaction of a block-level safety requirement. An example slice generated by our tool was already shown in Figure 8 and is not repeated here.

7.3. Inspection Assistant

The Inspection Assistant feature aims to help users better manage the inspection process. In particular, Inspection Assistant can record the decisions made during (1) inspections of atomic requirements (conducted over design slices), and (2) inspections done to ensure that the atomic requirements together lead to the satisfaction of higher-level requirements and ultimately the system-level safety requirements.

Figure 11 shows the various states an atomic requirement can go through during the inspection process. The initial state is “Yet to approve”. The user then reviews the design and marks the requirement as “Approved” or “Not approved”, depending on whether s/he deems the design as satisfying the requirement or not. Because the design evolves over time, it is important to check the impact of design changes on safety requirements. Based on the traceability information, Inspection Assistant detects the requirements that need to be re-inspected due to changes. Specifically, if there is a change made to any of the block operations or attributes

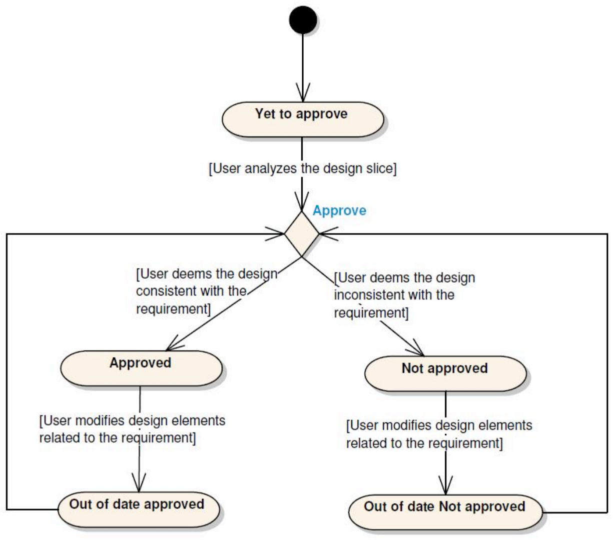

Yet to approve
[User analyzes the design slice]
Approve
[User deems the design consistent with the requirement]
[User deems the design inconsistent with the requirement]
Approved
Not approved
[User modifies design elements related to the requirement]
[User modifies design elements related to the requirement]
Out of date approved
Out of date Not approved

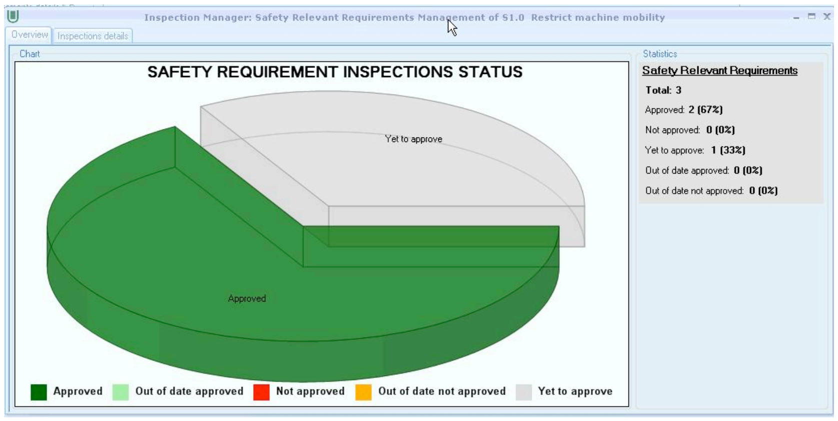
Figure 11: Inspection states for an atomic requirement

to which an atomic requirement is traced, then that requirement needs to be re-inspected. The status of an atomic requirement that needs to be re-inspected is set to “Out of date”; the status of the related higher-level requirements is set to “Out of date” as well. In addition, any change to the textual description of a requirement (at any level) would render the state of that requirement “Out of date” and the state will be propagated up to all the higher-level requirements.

To facilitate monitoring the progress of the inspection activities, Inspection Assistant can generate pie charts to visualize the relative proportion of requirements in different states. An example is shown in Figure 12. This pie chart depicts the status of the safety-relevant requirements that contribute to a selected system-level requirement (intermediate block-level requirements were filtered in this chart). The chart indicates that there are three safety-relevant requirements for the given system-level requirement and out of these, two have been already approved and the third is awaiting inspection.

7.3.1. Report Generator

Report generation is an important feature for supporting safety inspections. Reports are useful, for example, when a printed document needs to be signed off for legal obligations or simply for sequential reading of the inspection material. SafeSlice supports the automated generation of reports in PDF format. In particular, the tool allows the user to select the information to include in the report (e.g.


Inspection Manager: Safety Relevant Requirements Management of S1.0 Restrict machine mobility
Overview
Inspections details
Chart
SAFETY REQUIREMENT INSPECTIONS STATUS
Yet to approve
Approved
🟩 Approved
🟩 Out of date approved
🟥 Not approved
🟧 Out of date not approved
⬜ Yet to approve
Statistics

Safety Relevant Requirements

Total: 3

Approved: 2 (67%)

Not approved: 0 (0%)

Yet to approve: 1 (33%)

Out of date approved: 0 (0%)

Out of date not approved: 0 (0%)

Figure 12: Monitoring the inspection progress using pie charts

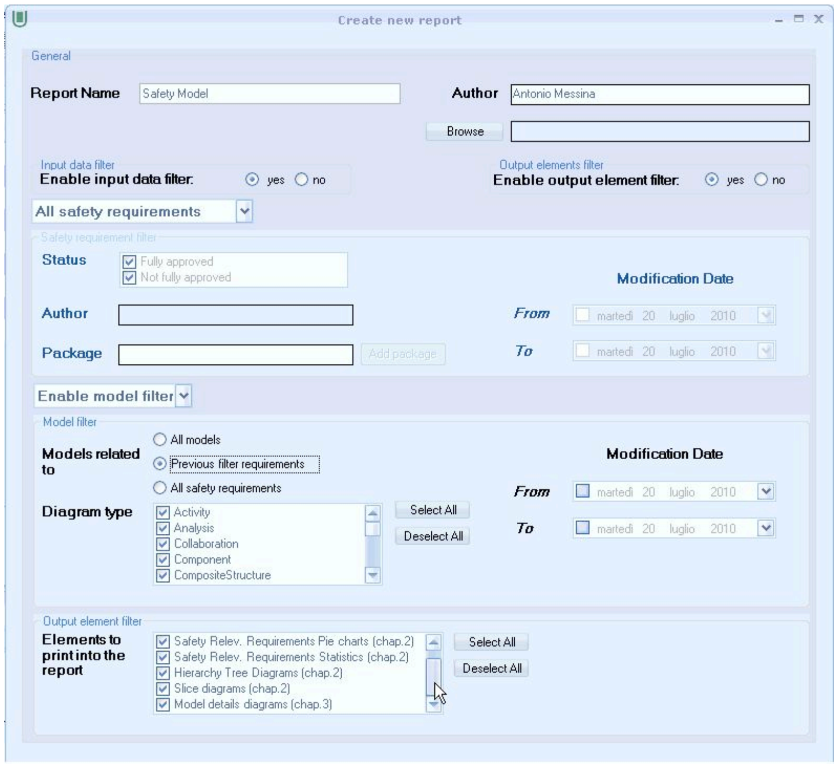
design slice, whole design, statistics, pie charts). This is shown in the snapshot of Figure 13. For instance, it is possible to generate a document reporting the inspection details of a given requirement or one that reports all the inspection details of all the requirements in a project.

7.3.2. Advanced Navigator

To make the inspection process more effective, SafeSlice supports advanced search and navigation of model elements. The Advanced Navigator feature allows the user to select an element x (e.g., a requirement, class, block, etc.) and retrieve the elements of given types (e.g., a specific requirements, classes, blocks, etc.) that have a given type of relationship with x (e.g., satisfy, trace, derive, etc.). Figure 14 shows a screenshot of the advanced navigation search box. The user specifies the element/relation types to search for from a pre-defined list. This list can be modified by the user if needed, e.g., when a new SysML stereotype is defined for capturing new types of elements.

8. Case Studies and Lessons Learned

To validate the feasibility and usefulness of our methodology (Section 6), we have conducted two case studies: the first case study is the Production Cell System, a small fragment of which was introduced in Section 4 as the running example for this article; the second case study is a real-world industrial system from the maritime and energy domain. Below, we first describe each of these two case studies (Section 8.1), and then report on the experience gained from applying our methodology to the case studies (Section 8.2).


Create new report

General

Report Name: Safety Model
Author: Antonio Messina
Browse

Input data filter
Enable input data filter: [✓ yes] no

Output elements filter
Enable output element filter: [✓ yes] no

All safety requirements

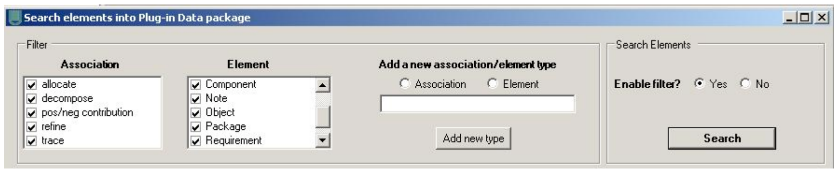
Safety requirement filter
Status: ✓ Fully approved, ✓ Not fully approved
Author:
Package: Add package
Modification Date
From: martedì 20 luglio 2010
To: martedì 20 luglio 2010

Enable model filter

Model filter
Models related to: All models [✓ Previous filter requirements] All safety requirements
Diagram type: ✓ Activity, ✓ Analysis, ✓ Collaboration, ✓ Component, ✓ CompositeStructure
Select All
Deselect All
Modification Date
From: martedì 20 luglio 2010
To: martedì 20 luglio 2010

Output element filter
Elements to print into the report: ✓ Safety Relev. Requirements Pie charts (chap.2), ✓ Safety Relev. Requirements Statistics (chap.2), ✓ Hierarchy Tree Diagrams (chap.2), ✓ Slice diagrams (chap.2), ✓ Model details diagrams (chap.3)
Select All
Deselect All

Figure 13: Screenshot of Automated Report Generation. The user can select the information to include in the report to be generated.


Search elements into Plug-in Data package

Filter
Association: ✓ allocate, ✓ decompose, ✓ pos/neg contribution, ✓ refine, ✓ trace
Element: ✓ Component, ✓ Note, ✓ Object, ✓ Package, ✓ Requirement
Add a new association/element type: Association Element
Add new type

Search Elements
Enable filter? [✓ Yes] No
Search

Figure 14: A screenshot of the advanced navigation feature of SafeSlice. The user can select the specific types of elements having a specific types of relations with the current element.

8.1. Description of the Case Studies

8.1.1. Benchmark Case Study

In our first case study, we applied our methodology to the Production Cell System (PCS), described earlier in Section 4. PCS is a well-known exemplar in system engineering and has been previously used as a benchmark to evaluate the capabilities of various specification methods for the purpose of safety analysis and verification [9]. The goal of this case study was to have an initial validation of the methodology before applying it to a real industrial system (see Secion 8.1.2). A complete SysML model of the PCS was developed based on the methodology proposed in this article. The model built for the PCS includes all the SysML diagrams in our methodology. In particular, the PCS design consists of 58 diagrams, 479 elements having 419 relations and 189 attributes. The PCS specification has 10 safety requirements, for which 10 design slices were generated using our slicing algorithm.

8.1.2. Industrial Case Study

Our industrial 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. Our main criterion in choosing this particular IO module from the several candidate IO modules was representativeness. The decision was made by the lead engineer of the IO modules at the company where we conducted our study, as she deemed the structure and the behavior of the selected IO module to be representative of the significant majority of the IO modules developed at the company.

We applied our methodology to the IO module under study and developed a complete SysML design with traceability to the module's requirements. To develop the SysML design and the requirements traceability links, we relied on interviews with the developers and an analysis of the existing documentation and source code. The resulting design and traceability links were iteratively validated and refined in collaboration with the lead engineer of the IO modules.

Since the IO modules at the partner company are used in safety monitoring and control systems, most of the modules' requirements are *safety-relevant*. As we stated earlier, this means that their requirements in some way contribute to the satisfaction of the system-level safety goals. An example of safety relevant requirement is "It shall be possible to manually trigger data transmission".

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

8.2. Lessons Learned

In this section, we discuss the lessons learned from applying our methodology to the two case studies described in Section 8.1.

Use of the SysML Language. Overall, we found SysML to be a good fit for capturing the behavioural 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 3. 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.

Level of Required Effort. The effort required by the 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 required effort is manageable, 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.

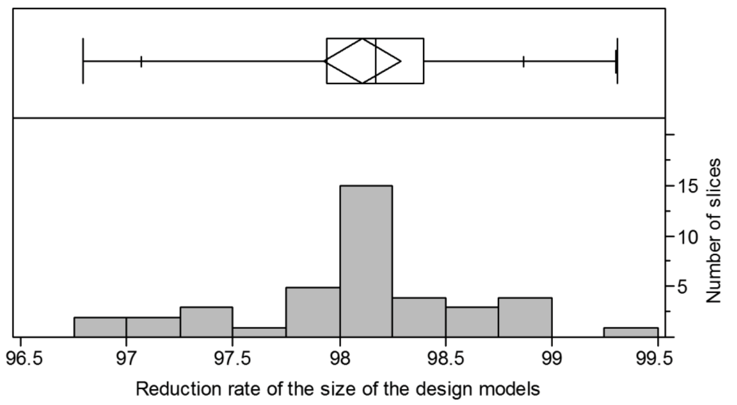


Figure 15: Frequency distribution and quantile box for slicing reduction

Usefulness of Slicing. The slicing procedure significantly 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 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 15 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 small, considering that the two systems are in different domains, and were modelled by different people. We therefore anticipate reduction rates to be close to the values observed here for other systems modelled according to our methodology.

In Section 6.2, 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. For whether a slice provides complete information to conduct the task the slice is intended for, as we already argued in 6.2, 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 included in the slices that will be generated in the future.

Slicing Performance. Our tool, SafeSlice, requires a negligible amount of execution time. We recorded the time required by SafeSlice to produce a design slice and to check compliance to the traceability information model in Section 5. For the larger of the two systems in our studies (i.e., PCS), SafeSlice had a worst-case time of ten seconds to produce a design slice and took about half a minute to check the compliance of the entire system to the traceability information model in Figure 6.

9. Related Work

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. [18] apply information retrieval techniques to find candidate links between development artifacts; Egyed [19] utilizes run-time information to suggest links between the system's models and the system's implementation; and Jirapanthong and Zisman [20] 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 [21] provide such an information model based on an observation of the practices in several software organizations, and Panesar et al. [22] – 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 [23], 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 paper 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.

Slicing techniques have been studied for a long time as means to reduce the complexity of software development. Most of the existing work on slicing is concerned with program code, where slicing is mainly used as a debugging aid [24]. Various other applications for code slicing have been described in the literature including program comprehension, software maintenance, and testing (see [25, 26] 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. [27] provide a technique for the slicing of state-based models using dependence analysis, and Kagdi et al. [28] 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.

Our concept of design slice is similar in theme and aim to architectural views [29]. 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 [29, 30, 31].

10. Conclusion

In this paper, we developed a traceability framework to facilitate the software 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 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 serious 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 leading SysML modeling environment. The tool guides the construction of traceability links, maintains consistency between the links and the information model, and provides facilities for automatic slicing.

Our work in this paper 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, the developers are left without proper guidelines on how to build and relate their models in a way that is suitable for the analyses they intend to perform over the models, e.g., certification, impact analysis, automated testing and verification. As a result, they often do not model the right aspects of the system or miss crucial details. This problem must be addressed through the development of more concrete and validated methodologies. 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 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.

Future work will include the development of templates for writing requirements so that requirements can be more systematically mapped onto design elements, e.g., block operations and states. Ultimately, we plan to conduct larger industrial case studies to assess the extent to which developers benefit from our framework so as to obtain a more conclusive picture of the cost-benefit trade-offs for traceability in the context of safety certification.

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

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

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

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

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

[10] 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).

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

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

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

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

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

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

[17] J. López-Grao, J. Merseguer, J. Campos, From uml activity diagrams to stochastic petri nets: application to software performance engineering, in: WOSP, 2004, pp. 25–36.

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

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

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

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

[22] 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.

[23] 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.

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

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

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

[27] 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.

[28] 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.

[29] Ieee std 1471:2000, now also iso/iec 42010:2010, systems and software engineeringÑarchitecture description (2000).

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

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