Traceability and SysML Design Slices to Support Safety Inspections: A Controlled Experiment

LIONEL BRIAND, University of Luxembourg, SnT centre
DAVIDE FALESSI, Fraunhofer USA, Center for Experimental Software Engineering
SHIVA NEJATI, University of Luxembourg, SnT centre
MEHRDAD SABETZADEH, University of Luxembourg, SnT centre
TAO YUE, Simula Research Laboratory

Certifying safety-critical software and ensuring its safety requires checking the conformance between safety requirements and design. Increasingly, the development of safety-critical software relies on modeling, and the System Modeling Language (SysML) is now commonly used in many industry sectors. Inspecting safety conformance by comparing design models against safety requirements requires safety inspectors to browse through large models and is consequently time consuming and error-prone. To address this, we have devised a mechanism to establish traceability between (functional) safety requirements and SysML design models 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. In this paper, we report on a controlled experiment assessing the impact of the traceability and slicing mechanism on inspectors' conformance decisions and effort. Results show a significant decrease in effort and an increase in decisions' correctness and level of certainty.

Index Terms— Empirical Software Engineering, Software and System Safety, Requirements Specification, Design, Software/Program Verification

1 INTRODUCTION

An important activity in the development of safety-critical systems is to ensure that the design of such systems meets the systems’ safety requirements. Design inspections (or reviews) are one of the main vehicles for ascertaining the satisfaction of safety requirements by the design [Falessi, et al. 2012b]. IEC 61508 [IEC 2005], a widely-used safety standard in the industry for embedded systems, recommends that a design review should be conducted for all new or modified (safety-critical) systems. The standard rates design reviews as Highly Recommended (the highest importance rating) for all systems at any level of criticality. In a recent literature survey, Nair et al [Nair, et al. 2013] found that results from requirements, design, and code inspections are regularly used as evidence during certification to justify the safe operation of a system.

With the use of model-driven development being on the rise, particularly in safety-critical application domains, engineers are more frequently faced with inspection tasks that involve models. For large projects, the models may be large and complex [Software Engineering Institute 2006]. Further, the number of models and model dependencies may be large, e.g., when multiple stakeholders and organizations are involved, or when the development effort is distributed over several sites.

A major problem that we observed while introducing model-driven development into industrial safety-critical systems was that large interrelated models needed to be inspected by people who had not been involved in the construction of the models [Nejati, et al. 2012]. Inspecting compliance between safety requirements and the design was, as a result, time consuming and forced the inspector to browse through the models and manually analyze large numbers of dependencies between requirements and models, and within the models themselves. Such inspections are typically performed for each version of the system, where changes must also be shown not to have any safety implications.

This paper presents a controlled experiment aimed at assessing a traceability-based mechanism for slicing of system design models. The goal of the slicing mechanism is to help analysts narrow the focus of their work when they need to inspect whether the system design meets a given requirement. Such support is typically
needed in safety-critical systems when safety inspectors or certification bodies must provide assurance regarding the safety of a system.

We use the Systems Modeling Language (SysML) [OMG 2008] for modeling system designs. SysML is increasingly used for modeling safety-critical applications since safety is a system concern and not only a software concern. Therefore, the relationship between software and the other components in a system is crucial for assessing software safety concerns. The slicing mechanism being assessed here is based on defining careful traceability mappings between functional (or behavioural) safety requirements and SysML design components. We use these mappings for extraction of design slices (model fragments) that filter out irrelevant design details but keep enough context information for the slices to be easy to understand and inspect. Traceability between requirements and design is further required by many industry standards for safety-critical systems.

The controlled experiment in this paper builds on our earlier work, [Nejati, et al. 2012], where we presented our technical solution for design slicing and the mathematical properties of design slices. In this earlier work, we further described tool support and demonstrated via case studies firstly, that our approach is a suitable match for modeling industrial systems, and secondly, that model construction and traceability as envisaged in our approach are feasible in terms of costs. These earlier case studies were a necessary first step towards the evaluation of our approach, particularly to ensure that the approach would have the potential for adoption in a realistic setting.

This paper attempts to assess, in a controlled and quantitative fashion, the impact of our proposed traceability and slicing mechanism on the effort and effectiveness of safety inspections, that is, inspections checking compliance between safety requirements and design decisions expressed as SysML models. The experiment controls for possible confounding effects of subjects’ skills, learning and fatigue effects, and design defect types. Results clearly show benefits in terms of increasing the correctness of conformance decisions (from 50% to 63%), decreasing the proportion of uncertain decisions (-45%), and reducing the effort of inspections (-27%).

These results complement the case studies in [Nejati, et al. 2012]. More precisely, the earlier case studies demonstrated that our approach meets the minimum requirements in terms of expressiveness and costs to be applicable in practice, whereas the current experiment demonstrates that the approach, if applied, provides perceivable benefits during safety inspections.

The paper is structured as follows. Section 2 provides background information, including related work and the case study system used for the experiment. Section 3 presents the main principles of our methodology for traceability and model slicing. Section 4 defines our research questions and experiment design and procedures. Results are reported in Section 5 and threats to validity in Section 6. Conclusions are drawn in Section 7.

2 BACKGROUND

2.1 Related Work

Slicing. Slicing mechanisms have long been studied as a way to reduce the complexity of software development activities. The majority of the existing work on slicing deals with program code. In this context, slicing is mainly used as an aid for debugging. This usage is motivated by the observation that a slice corresponds to the cognitive abstractions that developers make when they are debugging a program [Weiser 1984]. Numerous other applications for program slicing have been noted in the literature including program comprehension, software maintenance, and testing (see [Binkley and Gallagher 1996, Tip 1994] for surveys). Program slicing has more recently been used as a tool for reducing the complexity of automated verification as well. For example, the work in [Dwyer, et al. 2006] reports significant benefits from
using slicing as an automatic reduction technique for model checking.

In model-based development, slicing has been studied as a tool for reducing cognitive load and to facilitate understanding, inspection and modification of models. Various model slicing mechanisms exist: among others, in [Korel, et al. 2003] the authors provide a technique for the slicing of state-based models using dependence analysis, and in [Kagdi, et al. 2005] a technique is proposed for the slicing of UML class models based on predicates defined over the model’s features. Unlike our approach (Section 3), these approaches do not provide means to express the relationships between the requirements and the design models and hence cannot be used for extracting model slices with respect to a given requirement.

There is already a large body of empirical evidence showing the usefulness of slices for program analysis [Binkley and Harman 2003]; however, little is known about whether these benefits translate to model slices as well. In particular, we are not aware of any empirical study, similar to the present one, where the usefulness of model slices for inspections is investigated.

**Traceability.** The generation of model slices in our work makes use of traceability links between the requirements and the design. In general, traceability information can be used for a variety of verification and validation tasks, e.g., impact analysis or consistency checking between high-level textual requirements and use cases [Knethen and Grund 2003], among requirements [Falessi, et al. 2013], between requirements and design models [Knethen 2002, Letelier 2002], between UML models [Briand, et al. 2009b, Briand, et al. 2006], and between design elements and design rationale [Falessi, et al. 2006a]. Traceability further provides useful support mechanisms for managing requirements during the ongoing change process [Cleland-Huang, et al. 2005].

As for system and software inspections, traceability is a prerequisite for ensuring that the requirements have been correctly allocated and realized by the design or the source code components [Haumer, et al. 2000]. If the requirements, design, and implementation artifacts are regarded as different viewpoints on a system, establishing traceability is then the problem of building appropriate relationships between different viewpoints [Nuseibeh, et al. 2003]. The specification of these relationships can be systematized and guided by a traceability information model [Ramesh and Jarke 2001], which details the traceability information to be recorded and how this information should be structured. Substantial work has been done over the years to provide automated assistance in the retrieval of traceability information [Hayes and Dekhtyar 2005].

Results from existing empirical studies [Knethen 2002] suggest that high-quality traceability information has beneficial effects on the effectiveness and efficiency of understanding change, performing impact analysis, and evolving existing artifacts. However, to the best of our knowledge, there is no empirical study that evaluates the impact of requirements-design traceability on inspections, and especially so in the context of safety requirements.

An important concern in relation to traceability is cost-effectiveness. Making traceability cost-effective requires a careful analysis of the trade-offs between the costs incurred over establishing and maintaining traceability links and the benefits that traceability offers [Biffl, et al. 2006, Eyed, et al. 2005, Falessi, et al. 2012a, Falessi, et al. 2008, Heindl and Biffl 2005]. Traceability is considered worthwhile if it presents a significant advantage for achieving certain goals. In our case, the goals pursued from traceability are to increase the correctness and decrease the effort associated with design safety inspections. The level of improvement that would be deemed significant for these goals is a context-dependent issue and has to be determined based on an economic analysis in the domain of application. For example, a 5% decrease in effort might be considered significant enough to warrant establishing and maintaining traceability in a domain where inspections are very expensive to run; whereas, in another domain, a larger decrease of effort (or increase in correctness)
may be required before traceability pays off. In this paper, we take a conservative approach and consider traceability worthwhile if it improves correctness and decreases effort by at least 20%. This assumption will be later recalled in the experiment’s power analysis (Section 5.2.2) where the minimum effect size is 0.2.

**Design Rationale.** Reporting the rationale of design models [Burge, et al. 2008] can support several software architecture related activities like architectural reviews, reviews for a specific concern, and change impact analysis [Krupchen, et al. 2006]. Falessi et al. [Falessi, et al. 2006b] enacted a controlled experiment to study the impact of design rationale documentation on the correctness of and the required effort for assessing the compliance of old design decisions to new requirements. Results indicate that the use of design rationale documentation improved correctness significantly and did not impact effort. The study in this current article is similar to the one in [Falessi, et al. 2006b] in the sense that they both investigate how additional documentation (slice or design rationale) helps inspectors in terms of correctness and effort in checking design compliance. However, the objectives, artifacts, and context in [Falessi, et al. 2006b] are different from the present study.

**Architecture Views.** Our concept of design slices is similar in theme and aim to architectural views [Hofmeister, et al. 1999, IEEE 2000, Rozanski and Woods 2005]. In fact, both slices and architectural views capture a fragment of the whole design, consisting of several diagrams types (e.g., class diagrams and activity diagrams). In both cases, the fragment is extracted with the objective of reducing complexity and hence the human load for analysis and review. However, in the software architecture domain, 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 stakeholders concern [IEEE 2000, Lago, et al. 2010] which, in contrast to our case, represent higher-level goals, including non-technical aspects such as social, psychological, and managerial issues [Clements, et al. 2002, Kazman and Bass 2002].

### 2.2 Case Study System

The underlying system in our experiment is the Production Cell System (PCS). PCS is a well-known exemplar for embedded systems and has been previously used as a benchmark to evaluate the capabilities of various specification methods for the purpose of safety analysis and verification of embedded software systems [Lewerentz and Lindner 1995].

Briefly, the function of PCS is to transform metal blanks into forged plates using a hydraulic press. The process begins with the system operator (a human agent) putting a blank on a conveyor belt, called the feed belt. The feed belt conveys the blank to a (moving) table. This table takes the blank to a position where a robot arm can take it. The arm puts the blank on the press. After the blank is pressed, the forged plate is transferred into a container using a separate robot arm, a separate conveyor belt, and a crane arm.

The methodology proposed in [Nejati, et al. 2012] has been used to model the PCS to support safety certification, resulting in a design specification along with traceability links from the system’s safety requirements to the design. This study includes 23 requirements, of which 21 are functional (or behavioural) safety or safety-relevant requirements. PCS has 6 major hardware devices. In the design, we created 6 high-level software blocks corresponding to these devices. Each high-level software block is decomposed into two to three lower-level blocks representing smaller parts of the devices. To capture the behaviours of these blocks, we created 20 activity diagrams including one overview diagram capturing the parallel behaviours of all the high-level blocks, and 19 smaller diagrams representing the sequence of steps inside the major activity nodes in the overview diagram. The overview diagram includes 9 parallel swim-lines. Each parallel section in the overview diagram and each lower-level activity diagram has on average 11 activity nodes and 14 edges. We also created 7 state
machine diagrams. Each state machine diagram has on average 5 parallel sections; each section has on average 8 states and 15 transitions. The complete SysML requirements and design specification for PCS is available in [Briand et al. 2009a, Klykken 2009].

In addition to PCS, in our earlier work, we applied our methodology to a case study from the maritime and energy domain [Sabetzadeh et al. 2011]. The case study involved a safety input/output driver that was about to undergo certification as a standalone component. This driver, which is of similar complexity to the PCS, include a total of 30 functional requirements and 5 non-functional requirements (of which we focused on the functional requirements for creating traceability links and slices [Sabetzadeh, et al. 2011]). While PCS is a benchmark case study, its size and complexity are comparable to those of our real-world case study.

3 SLICING MECHANISM TO SUPPORT SAFETY INSPECTIONS

The main task of the safety inspectors during design inspections is to determine whether the safety requirements of a system are properly addressed by the design. To this end, it is important to be able to delineate the slice of the design that is relevant to each safety requirement.

Slicing enables safety inspectors to narrow the scope of their analysis to the smaller fragments of the design related to the task at hand. This is expected to reduce cognitive load and thus make it less likely that serious safety issues would be overlooked.

In [Nejati, et al. 2012], we have developed a SysML-based methodology that enables establishing the traceability links needed for slicing. Utilizing these links, we proposed an algorithm that automatically extracts the design slice relevant to a given requirement. The work in [Nejati, et al. 2012] concentrates specifically on functional (or behavioural) safety requirements, i.e., safety requirements that constrain some system behaviours. Non-functional requirements and cross-cutting concerns cannot typically be traced to particular design fragments, and as such, our work does not apply to them. For example, performance requirements are non-functional requirements that can affect system safety goals. Since these requirements constrain the behaviour of the system as a whole, they cannot be inspected by focusing on a particular design slice.

Although providing a detailed description of our design methodology and slicing algorithm is outside the scope of this current work (see [Nejati, et al. 2012]), it is important to provide a general understanding of the notion of slice, so that the motivation and the results that we present in this paper can be fully understood. To achieve this, we briefly discuss an information model that captures the traceability links underlying our methodology in Section 3.1. We then present, in Section 3.2, our algorithm for automatically extracting slices of the design diagrams relevant to a particular safety requirement. In Section 3.3, we discuss the criteria that design slices should meet to be effectively used by certifiers for inspecting and verifying safety requirements. Finally, in Section 3.4, we present the tool support for our traceability methodology and slicing mechanism.

3.1 Traceability Information Model

To be able to extract slices effectively and automatically, it is essential that first traceability links be established from the safety requirements to the design. In [Nejati, et al. 2012], we provide methodological guidelines on how to capture such traceability links for the purpose of safety certification. These guidelines led to an information model shown in Appendix A. Specifically, the information model specifies the traceability links suggested by our methodology as well as the well-formedness criteria of the links. We use an example from the PCS (Section 2.2) to illustrate the requirement-to-design traceability links prescribed by this information model.

In the specification of PCS, several functional safety requirements were stated to
ensure the system’s safe operation. The one we use for illustration is the following:

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

The above requirement concerns all the blocks interacting within PCS. In our information model, we refer to such requirements as system-level requirements. System-level requirements need to be decomposed into finer-grained requirements that constrain the behaviour of a few specific blocks before they can be traced to the system’s design.

In Fig.1, we have shown one of the (several) requirements resulting from the decomposition of **R**, namely, **R1:** The feed belt conveys a blank to table if the table is in load position. We refer to this more specific requirement that concerns the blocks FeedBelt and Table as a block-level requirement. We trace block-level requirements to those design blocks of the system that play a role in the satisfaction of the requirement, in this example, FeedBelt and Table.

We annotate the traceability links from block-level requirements to blocks with expressions that capture how the textual phrases in the requirement map onto the block operations and states. These mapping expressions are indicated by the element Mapping in Appendix A. In [Nejati, et al. 2012], we provide templates for generating the mapping expressions.

We use three mapping expressions to augment the traceability links from **R1** to FeedBelt and Table:

- The mapping attached to the link from **R1** to FeedBelt states that the requirement phrase “feed belt conveys a blank” is realized by the operation feed_table() of the FeedBelt block. The if-and-only-if symbol (↔) means that the operation is necessary and sufficient to satisfy the requirement phrase.

- The two mappings attached to the link from **R1** to Table state that the requirement phrase “table is in load position” holds when go_load_position() or when initialize() is executed. In other words, the post-condition of go_load_position(), i.e., the state of the Table block after go_load_position(), and the post-condition of initialize(), i.e., the state of the Table block after initialize(), each can imply the requirement phrase.

In [Nejati, et al. 2012], we provide methodological guidelines to help engineers carry out the following steps: 1) decomposition of system-level requirements into block-level ones, 2) creating traceability links between block-level requirements and system blocks, and 3) augmenting the traceability links with mappings. Omission of traceability information or any inaccuracies in the traceability links or the mapping phrases would lead to generation of incomplete slices, i.e., slices that do not contain
sufficient information for safety analysis. In this situation, the engineer may need to revise or refine the traceability links and the mappings because otherwise he may not be able to conclusively determine just based on the information in the slice whether a given requirement is satisfied by the design or not. As we further discuss in Section 3.3, slices are always sound, implying that they do not contain any information that contradicts the original design, and hence, they do not mislead engineers into making wrong decisions.

3.2 Design Slicing Algorithm

Given a set of SysML diagrams augmented with traceability information conforming to the model in Appendix A, and given a particular block-level safety requirement $BR$, we describe an algorithm for extracting a design slice (i.e., a set of fragments of the SysML diagrams) that is relevant to $BR$. The detailed description of the algorithm, which was first presented in [Nejati, et al. 2012], is included in Appendix B. Here, we briefly describe the algorithm and illustrate it.

Fig. 2 presents a high-level description of the algorithm in Appendix B. The algorithm takes as input a block-level safety requirement $BR$ and a set of SysML design diagrams conforming to the information model in Appendix A. It generates (1) a block diagram slice, including the set of blocks and relations between blocks, (2) an activity diagram slice, including a set of activity partitions, and (3) a state machine diagram slice, including a set of states and transitions between states. These sets are constructed such that all the blocks, the block operations/states, and the relations between blocks in the block diagram slice, all the activity nodes/edges in the activity diagram slice, and all the states and transitions in the state machine diagram slice, directly or indirectly, contribute to the satisfaction of $BR$. We go through the steps of the algorithm over the example in Fig. 1.

**Algorithm. GenerateSlice**

**Input:** Block-Level Requirement $BR$, and a set of SysML design diagrams conforming to the information model in Appendix A.

**Output:** Design slices related to $BR$.

**Step 1.** Identify design elements directly or indirectly related to $BR$.

**Step 2.** Extract block diagram slices:

- **Step 2.1** Remove operations, and attributes that are not related to $BR$ (directly or indirectly).
- **Step 2.2** Include relations between the blocks directly related to $BR$.

**Step 3.** Extract activity diagram slices:

- **Step 3.1** Identify activity partitions for the blocks directly related to $BR$.
- **Step 3.2** Remove activity nodes and edges not related to $BR$ (directly or indirectly) from these partitions.
- **Step 3.3** Add stuttering edges to maintain the connectivity between the diagram nodes.
- **Step 3.4** Identify new initial activity nodes.

**Step 4.** Extract state machine diagram slices:

- **Step 4.1** Identify state machines for the blocks directly related to $BR$.
- **Step 4.2** Remove states and transitions not related to $BR$ (directly or indirectly).
- **Step 4.3** Add stuttering transitions to maintain the connectivity between the states.
- **Step 4.4** Identify new initial states.

Fig. 2 Overview of the GenerateSlice algorithm.
The first step of the GenerateSlice algorithm identifies the design elements that are either directly mapped to BR, or are related to the elements directly mapped to BR. For example, consider the activity diagrams in Fig.3, specifying the behavior of FeedBelt and Table. Based on the traceability information provided by the engineer, the activity feed_table, and the transitions from go_load_position and initialize to sending the signal FeedTable are directly related to R1. These elements are highlighted with thick lines in Fig.3. The transitions from go_load_position and initialize to sending the signal FeedTable correspond to the post-conditions of go_load_position and initialize, respectively. In addition to these three elements that are directly related to R1, we include the following elements: The ending points of the transitions highlighted with thick lines in Fig.3, and the nodes related to sending the signals FeedTable and Go_Unload_Position. These additional elements are relevant to R1 because they either provide context information for the highlighted elements, or trigger (or are triggered by) the highlighted elements.

After identifying the set of design elements directly or indirectly related to BR in Step 1, Steps 2 to 4 compute the slices of block diagrams, activity diagrams, and state machine diagrams, respectively. For example, the block diagram and activity diagram slices for R1 are shown in Fig.4, and the state machine diagram slice for this requirement is shown in Appendix C. These diagrams include all the elements computed in Step 1. In addition, for activity and state machine diagrams, we need to ensure the following points: (1) The connectivity and ordering between activity nodes and between states must be preserved after the removal of edges. Hence, we add special transitions, called stuttering transitions, between the activity nodes and between states whose connecting paths are removed. These transitions are meant to preserve only the reachability relations between nodes and not the exact number of steps to go from one node to another [Abadi and Lamport 1991]. These transitions are shown using dashed arrows in Fig.4 and in Fig. 18 (Appendix C). (2) When the initial activity
nodes and states of the original diagrams are removed, we identify new initial activity nodes and states to maintain the entry points for the diagrams.

Fig. 4 The block and activity slices for R1 in Fig. 1. The state machine slice for R1 is shown in Appendix C.

As we discuss in Section 3.3, adding stuttering edges and transitions, and identifying initial nodes of activity diagram and state machine diagram slices enables us to keep the temporal orderings of the nodes in the slices consistent with those of the nodes in the original diagrams, hence ensuring the soundness of our algorithm.

3.3 Properties of Design Slices

Ideally, the design slices generated by our algorithm should possess the following three properties in order to be effectively used by safety inspectors 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.

Minimality. The slice should not include any element that is not required for maintaining its soundness or its completeness. That is, a slice is not minimal, if after removing some of its elements, it is still sound and complete. We have proven that the design slices generated by our algorithm in Fig. 2 are always sound, however they may not necessarily be complete or minimal.

We explain and illustrate the above properties using the activity diagram slice in Fig.4 (b). Note that this particular design slice possesses all of the above properties: (1) Soundness: R1 (in Fig.1) requires go_load_position or initialize to occur before feed_table to ensure that the table is in the desired position prior to the execution of feed_table. Since the ordering between the sending of signal FeedTable and the go_load_position and initialize activities, and the ordering between the receiving of the FeedTable signal and the feed_table activity in the activity diagram slice in Fig.4 (b) are the same as the orderings between these nodes in Fig.3, the slice in Fig.4 (b) is sound for analyzing R1. (2) Completeness: The slice in Fig.4 (b) contains enough information to check the satisfaction of R1. To do so, we need to demonstrate that feed_table never happens if the table is not in its load position. In our example, this
translates into showing that \texttt{feed\_table()} in \texttt{FeedBelt} cannot occur unless 
\texttt{go\_load\_position()} or \texttt{initialize()} in Table have already happened. The activity slice in Fig.4 shows this is the case, i.e., \texttt{feed\_table}, can only occur when it receives the signal \texttt{FeedTable}. This signal is sent only after either \texttt{go\_load\_position} or \texttt{initialize} are executed. Note that this is under the assumption that the mappings in Fig.1 are complete, i.e., there is no operation other than \texttt{go\_load\_position()} or \texttt{initialize()} in PCS that may generate the \texttt{FeedTable} signal. Otherwise, if there were additional operations triggering the \texttt{FeedTable} signal, the engineer ought to provide traceability links and mapping expressions for them, and they would have appeared in the design slices. (3) Minimality: Removing any element from the slice in Fig.4 (b) would result in violation of the well-formedness of the slice diagrams, or the temporal consistency of the slices with the original diagram, or it would lead to loss of the context information required to understand and evaluate the slices.

In [Nejati, et al. 2012], we formally argue that the activity diagram and state machine diagram slices generated by the algorithm in Fig.2 are sound for temporal constraints. Intuitively, this is because the slices are created in such a way that the reachability relations between the nodes in the original activity diagrams are fully preserved in the slices.

In contrast to soundness, one cannot formally demonstrate that the generated design slices are complete or minimal. The completeness of a design slice depends on the completeness of the traceability links and the mappings attached to the traceability links. For example, if we remove from Fig.1 either of the mappings related to post(table.go\_load\_position()) or post(table.initialize()), the resulting activity diagram slice will not include the activity nodes go\_load\_position and initialize, respectively. Such omissions and errors do not affect soundness and are caught when the generated slices do not have sufficient information for analysis. A similar argument applies to the minimality criterion.

Our proposed traceability methodology, though useful for many purposes (e.g., impact analysis of requirements changes), must be justified by significant gains in the verification of compliance between (safety) requirements and SysML designs. The next sections of this paper report on a controlled experiment to assess the impact of our traceability and slicing mechanism. Note that we focus on the combination of the traceability methodology and slicing since they are two components forming a complete solution and cannot be studied separately.

### 3.4 Tool support

Our traceability methodology (Section 3.1) and slicing algorithm (Section 3.2) are implemented in a tool, named SafeSlice [Falessi, et al. 2011b]. The SafeSlice tool enables users to specify the traceability links envisaged by a given traceability information model, check the consistency of the established links, automatically extract slices of design with respect to functional safety requirements, and manage the progression status of the design inspection process.

SafeSlice is implemented as a plug-in for the Enterprise Architect tool (http://www.sparxsystems.com/ea) which, like other tools with support for SysML, has an environment for building SysML models and creating SysML’s built-in traceability links. The choice of developing a plug-in instead of a standalone application was made to seamlessly introduce SafeSlice into real development settings and further to simplify tool evolution and maintenance. Among the possible alternatives for modeling environments, Enterprise Architect was chosen due to its usability, wide industrial adoption, availability of detailed guidelines for plugin construction, and built-in support for storing and linking heterogeneous development artifacts (e.g., natural language requirements specifications, UML/SysML models, Word documents, source code).

Fig.5 shows the architecture of the SafeSlice tool. It communicates asynchronously with EA via events. All the information related to a development project is stored by
EA in a database. The plug-in 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 thus simplifies the implementation by hiding the underlying database technology.

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.

SafeSlice was publicly released in May 2011. For further information about SafeSlice, including a video demo, the source and installable files, visit: http://sites.google.com/a/simula.no/safeslice/.

4 EXPERIMENT PLANNING

4.1 Goal, Research Questions and Hypotheses

Our experiment objective can be formulated as a GQM goal [Basili, et al. 1994]: Analyze our SysML design slicing mechanism (Section 3) for the purpose of evaluation with respect to its impact on the feasibility, correctness, uncertainty, and required effort of inspections of requirement-design compliance, from the point of view of the safety inspector in the context of safety certification. Hypotheses are expressed using the notation $H_{ijk}$ with: $i$ indicating the dependent variable, $j$ the specific metric employed to capture the dependent variable, and $k$ the hypothesis type (0 if null, 1 if alternative).

The experiment objective above leads to the following specific research questions and hypotheses:

R.Q.1: Does the use of slicing improve the correctness of safety inspections?

Inspecting the compliance of design decisions with safety requirements is a difficult and time-consuming endeavour [Fallessi, et al. 2006b]. This is particularly acute when safety inspectors have not been involved in the original development and when the number of requirements and design components are large. We expect that the slicing mechanisms, by helping focus the inspection on relevant parts of the SysML design, will increase the ability of inspectors to identify inconsistencies. R.Q.1 leads to testing the following null hypothesis in our experiment.
The use of slicing does not have a significant impact on the correctness of safety inspections. The parameter \( i \) ranges from 1 to 6 to denote the six metrics considered to measure correctness (see Section 4.5.1.3).

\[ H_{1b} \]

R.Q.2: Does the use of slicing improve the feasibility of safety inspections?

Similar to correctness above, the expectation is that slices will help inspectors to form a clear opinion, independently of its correctness, on the compliance of design decisions with safety requirements. In our experiment, inspectors will have the possibility not to provide an answer if they do not feel sufficiently confident.

\[ H_{2b} \]

R.Q.3: Does the use of slicing reduce the uncertainty of safety inspections?

Similar to feasibility above, the expectation is that slices will help inspectors achieve higher confidence in their conclusions when checking the compliance of design decisions with safety requirements. In our experiment, we will allow inspectors to grade their degree of confidence when making a statement about compliance.

\[ H_{3b} \]

R.Q.4: Does the use of slicing reduce the required effort of safety inspections?

Safety inspections are time consuming and checking whether it can be reduced through the design slicing proposed here is an important practical question. Slicing is expected to help the inspector focus their attention on relevant parts of the model and is therefore expected to save inspection time.

\[ H_{4b} \]

4.2 Participants

Our controlled experiment was conducted at the University of Rome, TorVergata. The subjects in the experiment were 20 students taking a graduate course\(^1\) in empirical software engineering at the Department of Informatics, Systems and Production engineering (DISP). The course is given in the final year of a two-year masters-level education program in Informatics and Computer Engineering. All the students had been widely exposed to requirements engineering and UML in previous bachelor and masters courses. They had also undertaken course projects using the UML notation and eliciting requirements at different levels of abstraction, as defined by the Rational Unified Process [Kruchten 2003]. About 30% of graduate student typically have industrial experience, though this experience is not strongly related to the experimental tasks. Therefore it is difficult to tell whether such experience is relevant.

Subjects were not in any way pressured to participate in the experiment. We clearly explained that their course grade would not be related to their presence or performance during the experiment. This is an approach we have successfully adopted over several years in other experiments [Falessi, et al. 2010, Falessi, et al. 2009, Falessi and Cantone 2006, Falessi, et al. 2007].

4.3 Material

4.3.1 Safety Requirements

Ten out of the PCS’s total of 21 safety requirements were selected for our study. The criterion for selection was to minimize the diagrammatic overlaps between the design slices of the selected requirements, and thus maximize the diversity of the aspects of the entire design that were made subject to inspections. The selected requirements covered all the six high-level blocks of the PCS, while referring to distinct structural elements and block behaviours. Of the ten selected requirements, two were used for the pilot study (Section 4.7.2) run before the experiment and the remaining

\(^1\)http://www.uniroma2.it/didattica/ISP0910/
eight were part of the controlled experiment. All the requirements used in the experiment and for the pilot study are provided in Appendix D for reference.

4.3.2 Slices

Traceability was established from each safety requirement down to the design diagrams. Traceability links specify which parts of the design contribute to the satisfaction of each requirement (Section 3). Based on the established traceability links between the safety requirements and the design diagrams, we followed the slicing mechanism presented in Section 3 and derived the slices for each safety requirement used in the experiment. We did this manually because the tool was not yet ready to use. However, we have checked that SafeSlice produces the same slices as the ones manually generated.

In our context, each requirement has a combination of three types of slices: block, activity, and state machine, which respectively correspond to parts of block definition, activity, and state machine diagrams that all together contribute to the satisfaction of that requirement. The complete set of slices that were used in the experiment and for the pilot is available at http://people.svv.lu/nejati/ExperimentMaterial/.

The use of slices, when compared to using only traces, leads to significant reductions in the number of elements to inspect; this is because slicing extracts only the aspects that are relevant for inspecting a particular requirement. Fig. 6 shows the reduction frequency distribution and quantile box plot for the set of (safety) requirements in our experiment. For each requirement, the reduction rate was calculated as the ratio of the number of elements in the slice over the sum of the number of elements in all diagrams related to the requirement (as identified by the traces). On average, the use of slices reduces the number of elements by 88%.

![Fig. 6 Frequency distribution and quantile box for size reduction when using slices compared to only traces.](image)

4.3.3 Design Defects in SysML

4.3.3.1 Proposed Classification

The ultimate goal of the study is to investigate how our slicing mechanism can facilitate safety inspections of SysML design models. To do so systematically, we need a classification of different types of design defects that can occur in SysML models. For this purpose, we developed a checklist of design defects for SysML block definition diagrams, activity diagrams, and state machine diagrams, which are the three types of diagrams used in our controlled experiment. Before developing the checklist, we consulted the IEEE Standard Classification for Software Anomalies [IEEE 1994], which provides a detailed account of different software anomalies. This standard is however aimed at source code and documentation and therefore, cannot be applied directly in our context. Another classification that we looked at prior to creating our checklist was the classification of defects in state machine diagrams [Binder 1999],
developed for the purpose of evaluating state machine-based test design strategies. Our checklist is similar in nature to the above two classifications, but it is derived directly from analyzing the SysML and UML meta-models [OMG 2008, OMG 2009]. This helps ensure that the checklist is complete, at the right level of abstraction, and fully relevant to the inspection tasks. To help the subjects be more systematic in their search for defects, the checklist was included in the answer sheet (Appendix F) as it is commonly the case with checklist-based inspections [Brykczynski 1999].

4.3.3.2 Types of Seeded Defects

In the absence of similar previous studies, there is little information we can draw on for selecting a defect seeding strategy. In choosing defect types to seed from the overall classification of SysML design defects (Section 4.3.3), we considered three criteria that common sense and our experience indicated would be important to increase the realism of inspections:

**Human participation.** For the purposes of our experiment, we are interested only in defects that are difficult to detect automatically through a static analysis of the models. Thus, the selected defects should not merely be well-formedness rule violations. For example, cyclic inheritance in a block definition diagram is a defect, but it can be conveniently detected using an automatic rule checker without any assistance from a human inspector. Such defects are not of interest for the inspections in our experiment.

**High subjects’ familiarity.** The selected defects should be reasonably familiar to the experimental subjects. We should therefore focus on types of defects they were trained to detect, to emulate a realistic situation where inspectors are experienced.

**Similarity with common defects in the domain.** The types of defects adopted in the experiment are similar to the ones most commonly observed in practice.

The nature of design defects breaking a safety requirement can be either structural or behavioural [van Lamsweerde 2008]. Structural defects are those that have to do with missing or incorrect block specifications or inter-block relationships. These defects are typically found in block definition diagrams. For example, requirement **R1** in Fig.1 imposes two structural constraints: (1) **FeedBelt** and **Table** blocks interact by **FeedBelt** passing a blank to **Table**, and (2) only one instance of each of these blocks participates in this interaction. The former constraint is addressed through the “interact” association between **Table** and **FeedBelt**, as shown in Fig.1, and the latter is reflected as the multiplicities on the association (“1” on the both sides). Errors related to the association or its multiplicities are considered to be structural defects.

Behavioural defects, in contrast, break the system behaviour; they can impact the sequences of: (1) operations one block performs in response to other blocks operations, or (2) the sequences of events/messages communicated between different blocks. These defects appear in behavioural diagrams, i.e., activity/state machine diagrams in our experiment. For example, requirement **R1** in Fig.1 indicates that for PCS to behave correctly, operation **feed_table()** of **FeedBelt** must occur before operation **go_load_position()** of **Table**. As discussed in Section 3, the activity diagram in Fig.3 satisfies this ordering constraint. Any change in this diagram that breaks this particular ordering constraint causes a behavioural defect by violating **R1**.

As we discuss in Section 4.6, we seed only four types of defects into the PCS models in order to prevent confounding factors due to variations in defect types. To maximize the variability of defects, we chose to include two structural and two behavioural defect types from the checklist in Appendix F. Specifically, the selected structural defect types are: **Incorrect navigation of an association** (S0) and **Incorrect multiplicity of an association** (S1). The selected behavioural defects types are: **Wrong temporal ordering of operations of an activity diagram** (B0) and **Incorrect effect on transitions of a state machine diagram** (B1). S0 and S1 were seeded into block definition diagrams. Each defect type was applied to four requirements and hence instantiated four times. This gives a total of 4 * 4 = 16 actual seeded defects (see Section 4.6). In
Appendix E, we show one instance of each of the defect types seeded into the PCS models.

In our experiment, we strived to select defect types that satisfy the three criteria we discussed above in this section. However, we acknowledge that such a process, though well thought-out and performed before the experiment took place, is inevitably subjective.

4.3.4 Design of Experiment Material

The overall design of the experiment material is shown in Fig. 7. During the experiment, each subject carries out eight inspection tasks. In each of these tasks, the subject analyzes whether a specific block-level safety requirement is satisfied by a specific design. Inspection tasks are structured activities and follow certain steps, as we detail in Section 4.4. Each subject is provided with a package containing the material for the tasks they need to perform. There are eight parts to a package with each part providing the input material for one inspection task. The input material for each task includes the safety requirement in question and the complete system design. A slice is present in the material only when an inspection task is to be assisted by slicing. The design to be inspected in each task is either correct, or seeded with one defect. The defect types used in the experiment were discussed earlier in Section 4.3.3.2. There are thus four possible configurations of task input material: (1) requirement + correct design + slice; (2) requirement + correct design; (3) requirement + defective design + slice, and (4) requirement + defective design.

Fig. 7 Overview of the design of the experiment material.

For distributing the experiment material among the subjects, a shared medium, in our case, a single DVD, is produced. This medium is passed from subject to subject at the beginning of the experiment, so that each subject can obtain their personal package. To ensure that all subjects work on the right package, all the packages are password-protected. Each subject is issued with a package-password enabling the subject to open his/her own package but no one else's. These passwords are provided
to the subjects by the researchers at the beginning of the experiment.

Each subject performs each of the eight inspection tasks in a certain order as specified in the experiment design (Section 4.6). To enforce this order, the material for each task is password-protected as well. The password for the first task is provided to each subject at the beginning, i.e., together with the password to open the package. The password for each subsequent task is provided to a subject only upon the completion of the current task. All passwords are provided on paper by the researchers at the right times.

The above password-encryption scheme enables fine-grained control over the execution of the experiment, in turn reducing validity threats. The scheme further simplifies the instructions given to the subjects about the experiment procedure: the subjects only need to know they will obtain one password to open their package and eight passwords, delivered to them one at a time, to open their tasks. No complications arose during the experiment as the result of password encryption. Doing the password encryption of course entailed additional work for the experiment designers, as a total of 160 passwords were involved. We believe that this additional effort is outweighed by the benefits that the password scheme brings in terms of validity.

An answer sheet was designed for participants to report their analysis results (Appendix F). The question “How is the design in respect to the requirement?” was evaluated on a four-point Likert scale [Likert 1932], representing the extent to which the requirement is perceived to be consistent with the design under analysis. Such scales are commonly used in social science studies [Babbie 2010]. Possible answers are: “Definitely consistent”, “Likely consistent”, “Likely inconsistent”, and “Definitely inconsistent”.

Moreover, subjects were allowed to answer “I don’t know” in cases where they felt they could not form an opinion at all. Providing an option to opt out of answering a question, rather than forcing subjects to choose an answer, is an important consideration in questionnaire design, both to mitigate bias and to allow subjects to make progress without getting stuck on any particular task [Kitchenham and Pfleeger 2002].

Note that with the “I don’t know” option aside, the remaining answer options can be viewed as the Cartesian product of two Likert scales: (Consistent, Inconsistent) and (Likely, Definitely). Using two different scales would have changed the format of the answer sheet but not its semantics. Therefore, we chose the most concise format, given the number of elements in the Cartesian product was small. This is a common practice in social sciences [Oppenheim 1992].

In the answer sheet, we additionally provided a checklist and asked the subjects to classify the defects they found using the checklist. This was mainly aimed at mitigating the possibility that subjects would not search systematically for defects or not consider all defect types. Using a checklist is also consistent with how inspections are conducted in an industrial setting, where practitioners know the possible types of defects they are looking for beforehand, either based on a checklist or through experience [Cantone, et al. 2003, Thelin, et al. 2003].

### 4.4 Steps of the Inspection Task

Subjects, interacting only with the researchers, went through the following preliminary steps:

- Mark their presence in a signup sheet.
- Receive password-protected experiment material (package).
- Receive the package password.

Then, for each of the eight tasks, subjects enacted the following steps:

1. Receive the answer sheet (Appendix F).
2. Receive the password for the material to be used in this specific task.
3. Unzip the task material using the received password.
4. Record the starting time on the answer sheet.
5. Analyze the compliance of the design with the requirement in the task.
6. Record the analysis results on the answer sheet.
7. Record the completion time on the answer sheet.
8. Hand out the answer sheet.
9. Repeat from step 1 until all the eight tasks are completed.

4.5 Variables

4.5.1 Dependent Variables

In Section 3.3, we discussed the general criteria expected to be met by design slices, namely, soundness, completeness, and minimality. As we describe in more detail throughout this section, our experiment includes four dependent variables: feasibility (subject’s ability to perform a given task), uncertainty (level of subject’s confidence), correctness (whether the subject’s answer is right or wrong), and effort (how long it takes for the subject to provide an answer). Soundness, completeness, and minimality of slices relate to these four dependent variables as follows:

If slices are sound, the subjects can accurately determine whether a design is consistent with a requirement. Further, soundness increases the subjects’ confidence in checking compliance of the design and the requirement. Therefore, soundness is related to correctness and uncertainty.

If slices are complete, the subjects can conclusively determine whether a design is consistent with a requirement. Moreover, given complete slices, the subjects can rely exclusively on the information provided in the slices and therefore not need to refer back to the original design. This in turn reduces the effort required from the subjects for checking requirements-design compliance. Therefore, completeness is related to uncertainty and effort.

If slices are minimal, the subjects require less effort to filter out the irrelevant information from the slices, and hence require less effort to carry out a given inspection task. In addition, minimal slices improve scalability by avoiding cognitive overload. Therefore, minimality is related to effort and feasibility.

4.5.1.1 Feasibility

Feasibility captures whether subjects were able to draw a conclusion regarding design compliance with requirements. We measured feasibility with a binary scale; “Unfeasible” in case subjects answered “I don’t know”, “Feasible” otherwise.

4.5.1.2 Uncertainty

Uncertainty represents the level of confidence perceived by the subject regarding his design compliance conclusion. We measured uncertainty with a binary scale; “Uncertain” in case subjects answered “I don’t know”, “Likely consistent”, or “Likely inconsistent”; “Certain” otherwise when subjects answered “Definitely consistent” or “Definitely inconsistent”.

4.5.1.3 Correctness

Correctness captures whether a subject accurately determines whether a design is consistent or not with a requirement. Based on the information collected in the answer sheet described above, there are, however, several ways this can be measured as described in the following subsections.

4.5.1.3.1 Accounting for Feasibility and Uncertainty

To adopt a binary measurement scale for correctness (i.e., correct vs. incorrect), we need to choose a strategy for dealing with feasibility and uncertainty. There are a number of possibilities to consider:

First, we can deem a decision “correct” if the subject selected the correct “likely” or “definitely” option on the Likert scale. For example, if a design is inconsistent, both “Likely inconsistent” and “Definitely inconsistent” answers will be deemed correct.
Alternatively, we can consider only the latter category (i.e., “definitely”) to be correct. “I don’t know” answers can be considered incorrect or a special case (i.e., neither correct nor incorrect). None of these options is a priori wrong or better. Therefore, we need to explore all the options to obtain a complete picture of the results.

More precisely, there are four alternative ways to define correctness in relation to feasibility and uncertainty. Table 1 shows these alternatives (columns 3-6) according to the actual design consistency (1st column) and the subject answer (2nd column). The second row of columns 3-6 describes if subject answers that are uncertain or unfeasible are taken into account (“Y”) or not (“N”).

To illustrate the different interpretations in Table 1, let us consider an example. Suppose that in a given task, a subject finds the design to be “Likely consistent” where the design is actually inconsistent. The first interpretation (column 3 in the table) allows for both uncertain and unfeasible answers. Thus, the (uncertain) answer provided by the subject is acceptable according to this interpretation. The answer is naturally deemed “Incorrect” because the subject is leaning towards finding the design as being consistent whereas the design is actually inconsistent. The second interpretation (column 4 in the table) discards unfeasible answers but, similar to the first interpretation, accepts uncertain ones. Hence, the subject’s (uncertain) answer of “Likely consistent” is acceptable according to this interpretation as well, with the correctness outcome being just the same as in the first interpretation. As for the third and fourth interpretations (columns 5 and 6), neither accepts uncertain answers; hence, the subject’s answer is simply discarded and not counted as a data point for analysis. From this example, it is easy to see how the four interpretations in Table 1 would treat unfeasible (“I don’t know”) answers. All four alternative interpretations of correctness are used for analysis.

4.5.1.3.2 Precision and Recall

An inspection is effective if it neither classifies too many consistent designs as inconsistent (false positives), nor does it classify too many inconsistent designs as consistent (false negatives). A good way to evaluate the effectiveness of inspection activities is through the standard precision and recall metrics [Manning, et al. 2008]:

**Precision:** measures the fraction of designs deemed inconsistent by a subject that are actually inconsistent, i.e., (true positive) / (false positive + true positive).

**Recall:** Recall measures the coverage of subjects’ answers and is defined as the fraction of all inconsistent designs that are found to be inconsistent by a subject; i.e., (true positive) / (false negative + true positive).

Among the different correctness measures (Section 4.5.1.3.1), to analyze the Precision and Recall of subjects in classifying inconsistent designs, we decided to use correctness measured by accounting for both unfeasible and uncertain answers (Table 1, column 3). This yields the largest number of observations and is perhaps the least controversial option as no observation is discarded. Consistent with this choice, the last column of Table 1 (column 7) provides a classification for Correctness in terms of False/True positives/negatives according to the actual consistency of design (column 1) and the subject answer (column 2).

<table>
<thead>
<tr>
<th>Actual design consistency</th>
<th>Subject Answer</th>
<th>Correctness</th>
<th>Correctness Classification for Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Uncertain: Y</td>
<td>Uncertain: Y</td>
</tr>
<tr>
<td>Inconsistent</td>
<td>Definitely consistent</td>
<td>Incorrect</td>
<td>Incorrect</td>
</tr>
<tr>
<td>Inconsistent</td>
<td>Likely consistent</td>
<td>Incorrect</td>
<td>Incorrect</td>
</tr>
<tr>
<td>Inconsistent</td>
<td>Definitely inconsistent</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>Inconsistent</td>
<td>I don’t know</td>
<td>Incorrect</td>
<td>-</td>
</tr>
<tr>
<td>Consistent</td>
<td>Definitely consistent</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>Consistent</td>
<td>Likely consistent</td>
<td>Correct</td>
<td>Correct</td>
</tr>
<tr>
<td>Consistent</td>
<td>Definitely inconsistent</td>
<td>Incorrect</td>
<td>Incorrect</td>
</tr>
<tr>
<td>Consistent</td>
<td>I don’t know</td>
<td>Incorrect</td>
<td>-</td>
</tr>
</tbody>
</table>
Regarding false positives and false negatives, we note that in the context of safety-critical systems, it is more important, for the most critical requirements and components, to ensure that actual defects are detected rather than avoid the detection of false positives. Therefore, in the context of safety-critical systems, a high Recall is highly desired and more important than a high Precision.

4.5.1.4 Required Effort

We measured effort in number of seconds required by subjects for assessing the compliance of a design with a requirement, i.e., Completion time – Starting time (see steps 4 and 7 in Section 4.4).

4.5.2 Independent Variables

The main factor we wanted to assess was the impact of the use of slices on the dependent variables (Section 4.5.1).

We can think of two potentially important secondary factors that need to be balanced in the experiment design and then analysed to determine whether they have a main or interaction effect with slicing, on the dependent variables.

Step. In order to observe any learning or fatigue effect, it is important to observe how the dependent variables vary according to the successive experiment steps (eight in our case). The subjects’ skills in using and inspecting slices may improve as they consider more requirements. As with every new technology, in order to be adopted, its learning curve should be reasonable. Therefore, assessing this learning curve for our safety inspections would be interesting in order to understand the applicability of the approach in practice. Moreover, the subjects’ performance in the later steps are perhaps more representative of the benefits that our approach can potentially provide in the long term. Conversely, sometimes fatigue effects are observed as subjects lose motivation and energy as the experiment proceeds.

Type of defects. As described in Section 4.3.3, we considered and seeded different types of defects. The ability of humans in finding defects could vary according to the nature of defects. For instance, behavioural defects may be more difficult to find than structural ones, because structural diagrams can be inspected locally by checking individual blocks and relationships between each block pair, while for behavioural diagrams, one needs to gain a global picture of the system behaviour by mentally simulating the temporal sequences of actions/events/messages. Conversely, structural defects might be difficult to detect when a change in a pairwise relation has an unforeseen side-effect on other blocks not participating directly in that relation. For example, adding a generalization relation between two blocks may introduce a cyclic inheritance between several blocks not directly related to those two blocks. Moreover, the slicing mechanism may be more precise for certain types of diagrams. All these conjectures clearly show why the effect of the type of defects is difficult to predict and so important to investigate.

4.6 Experiment Design

We adopt a randomized crossover design [Montgomery 2006] consisting of two rounds. In a randomized crossover design, all the treatments are applied several times to each subject in a way that balances confounding effects. Each round consists of four steps. In each step, subjects inspected the SysML design (consistent or not) with respect to a given requirement. The duration of each round was one hour; the duration of a single step was neither imposed nor suggested to the subjects. Subjects were randomly divided into four groups in order to ensure reasonable consistency in skills and experience. The material was the same within the same group and different across groups.

Table 2 describes in detail the experiment design. The first and second columns indicate the experiment rounds and the steps, respectively. The sub-columns under the overall Subject Group column indicate the four random subjects groups. Each of
these sub-columns is composed of four further sub-columns: R indicates the ID of the requirement; IC indicates the input consistency; i.e., the presence (0) or absence (1) of a defect in the design; TD indicates the type of defect seeded in the design, if any (see Section 4.3.3.2); and P indicates the presence (1) or absence (0) of a slice.

Table 2 Experiment design.

<table>
<thead>
<tr>
<th>Round</th>
<th>Step</th>
<th>Subjects Group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 2 3 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R IC TD P R IC TD P R IC TD P R IC TD P</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 1 NA 1 2 0 S0 0 6 1 NA 1 8 1 NA 0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1 2 2 NA 1 3 1 NA 0 8 0 B0 1 5 0 B0 0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3 0 S0 1 4 1 NA 0 5 1 NA 1 7 1 NA 0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4 0 B0 1 1 0 B1 0 7 0 S0 1 6 0 S1 0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>5 1 NA 0 7 1 NA 1 3 0 B1 0 4 1 NA 1</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7 0 B1 0 6 0 B0 1 4 0 S1 0 1 0 S0 1</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>6 1 NA 0 8 1 NA 1 1 1 NA 0 2 0 B1 1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8 0 S1 0 5 0 S1 1 2 1 NA 0 3 1 NA 1</td>
</tr>
</tbody>
</table>

In our case, the randomized crossover design helps avoid confounding effects of subjects, defect types and requirement characteristics on our dependent variables:

Subject characteristics: All subjects applied both treatments the same number of times. Groups were formed randomly and therefore subject skills and experience are deemed comparable across groups.

Defect types: Each of the four defect types is instantiated four times across the two levels of the independent variable (slicing or not). In other words, all subject groups worked on exactly one instance of each defect type. Using more defect types would mean that the number of instances of each defect type across independent variable levels or across requirements would not be balanced, hence leading to confounding effects. Because each defect type is applied once in each group with a different requirement, we never seed the same defect instance twice. This is because for each group, there are four steps where inconsistent designs are inspected. We then seed four defect instances for each defect type, one for each requirement it is applied to in each group. This yields a total of 16 defect instances.

Requirements characteristics: Both treatments were applied on all requirements the same number of times. We used different requirements for each of the eight steps.

Presence of defect: Both treatments were applied, the same number of times, in the presence and absence of a defect.

Moreover, as described in Section 4.5.2, the design balanced the following secondary factors:

Learning: Both treatments were applied across steps the same number of times.

Type of error: Both treatments were applied on structural and behavioural types of defects the same number of times.

4.7 Procedure

4.7.1 Training

The training of subjects was conducted by one of the authors. The training consisted of four sessions (two-hours per session) and its aim was to ensure that the subjects were competent enough to perform the tasks. Subjects already received substantial training on UML in previous university courses, therefore our training focused on teaching the differences between SysML and UML. How to check for design defects using examples was also discussed during the training. A general overview of the case study was presented in order to familiarize the subjects with the domain and terminology of the production cell system. The presence of subjects in all the training
sessions was mandatory for participating in the experiment. Three training sessions were scheduled in three consecutive weeks; one training session was scheduled after the pilot study presented next.

4.7.2 Pilot

The goal of the two-hour pilot study was twofold: (1) help subjects to familiarize themselves with the experimental task, and (2) help us assess if the experiment design and material were adequate. The pilot study was performed in the same manner as what we had in mind for the actual experiment, but in a smaller scale. The first 30 minutes of the pilot were dedicated to describing the tasks and Q&A. During the pilot execution, subjects inspected the compliance of the design with two requirements, including one requirement that was supplied with its related design slice. After the pilot study, during a specific training session, the requirements and the design were discussed; the defect, if any, was described and discussed. Subjects described their difficulties in finding the defect or in correctly assessing design compliance and not overlook defects. Subjects felt this training session was useful to prepare them for the upcoming experiment. Subjects perceived the material and the design to be adequate; therefore no change to the experiment material or experiment design was made.

4.7.3 Experiment Execution

Subjects used their own notebook computer for retrieving and reading the SysML designs. A large screen located in the experiment premises provided information about subjects’ ID and showed the experiment design as presented in columns Round, Step, R and P in Table 2. No subject was allowed to discuss his or her work with other subjects during the experiment.

Of the 20 invited students, 17 participated in the experiment. They all followed the prescribed procedures; no major difficulties were encountered throughout the execution of the experiment. The only minor problem was that three subjects, A, B, C, ran out of time. Subject A failed to finish two tasks, both of which involved slices. Subjects B and C failed to finish one task each; neither task involved slices. We therefore obtained a total of (14 * 8 + 6 + 7 + 7) = 132 observations (completed tasks). Of these, 66 involved slices and the other 66 involved no slices.

4.7.4 Data Collection

Because the data collected during the experiment was paper-based, we encoded data in a JMP® data file prior to analysis. The encoding activity took place just after the experiment execution and involved five persons, including four volunteer subjects and one of the authors. The encoding process was structured according to four different roles: dictating, encoding, checking the dictating, and checking the encoding. The encoding process lasted almost one hour and subjects randomly changed roles around every 10 minutes. The output of the encoding process was a JMP data file containing a table with the following information: ID of the subject, ID of the requirement, Step, Initial time, Answer, Final time.

Data integrity was checked using the following rules: for the same subjects and for each step, the Initial time must precede the Final time of the same step, and the final time must precede the Initial time of the next step. Moreover, the consistency of the tuple (subject ID, Requirement ID, Step) with what was prescribed in the experiment design was checked (Section 4.6).

5 RESULTS AND DISCUSSION

We analyze and present our experiment results in this section. The descriptive statistics for each research question are described in Section 5.1, followed by the statistical testing results in Section 5.2. We analyze the impact of the experiment steps and the type of defects on the dependent variables in Section 5.3. Last, we discuss the

http://www.jmp.com/
results in Section 5.4.

5.1 Descriptive Statistics

In this section, for each research question, we report descriptive statistics showing overall trends.

5.1.1 R.Q.1: Does slicing improve the correctness of safety inspections?

5.1.1.1 Feasibility and Uncertainty aspects

Table 3 reports the raw data results in terms of frequency of subjects’ answers. Given the definitions of Correctness in Table 1 and the raw data in Table 3 we computed Fig. 8; this describes the proportions of correct answers where the two different gray shades indicate the presence or the absence of slicing (two treatments). Correctness clearly improves with slicing regardless of the way we measure it, though to various degrees.

Table 3 Raw data results.

<table>
<thead>
<tr>
<th>Actual Design Consistency</th>
<th>Subject Answer</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>With Slicing</td>
</tr>
<tr>
<td>Consistent</td>
<td>Definitely Consistent</td>
<td>13</td>
</tr>
<tr>
<td>Consistent</td>
<td>Definitely Inconsistent</td>
<td>9</td>
</tr>
<tr>
<td>Consistent</td>
<td>Likely Consistent</td>
<td>6</td>
</tr>
<tr>
<td>Consistent</td>
<td>Likely Inconsistent</td>
<td>4</td>
</tr>
<tr>
<td>Consistent</td>
<td>I don’t know</td>
<td>0</td>
</tr>
<tr>
<td>Inconsistent</td>
<td>Definitely Consistent</td>
<td>5</td>
</tr>
<tr>
<td>Inconsistent</td>
<td>Definitely Inconsistent</td>
<td>22</td>
</tr>
<tr>
<td>Inconsistent</td>
<td>Likely Consistent</td>
<td>5</td>
</tr>
<tr>
<td>Inconsistent</td>
<td>Likely Inconsistent</td>
<td>1</td>
</tr>
<tr>
<td>Inconsistent</td>
<td>I don’t know</td>
<td>1</td>
</tr>
</tbody>
</table>

5.1.1.2 Precision and Recall

Fig. 9 reports the Precision and Recall in case of absence and presence of slicing. We can see that slicing significantly improves the Recall of correctness classification and, to a much lesser extent, its Precision. Slicing increases the probability of finding a design defect from 31% to 68% (+120%). In other words, slicing makes it much more unlikely to overlook a non-compliant design.

5.1.2 R.Q.2: Does Slicing Improve the Feasibility of Safety Inspections?

Feasibility is assessed through “I don’t know” answers (Section 4.5.1.1). As shown in Fig. 10, the use of slicing decreased the number of these answers from 10 to 1 (-90%), clearly helping subjects make decisions.

5.1.3 R.Q.3: Does Slicing Reduce the Uncertainty of Safety Inspections?

If we consider both “Likely” with “I don’t know” answers to be uncertain, the use of slicing decreases uncertainty from 47% to 26% (-45%), as shown in Fig. 11.

5.1.4 R.Q.4: Does Slicing Reduce the Required Effort of Safety Inspections?

The use of slicing decreases the average time required to check a requirement from 683 to 496 seconds (-27%), as shown in Fig.12.
5.2 Statistical tests

5.2.1 Hypothesis Testing

In this section, we test whether the differences observed in the previous section are statistically significant, i.e., whether our null hypotheses (Section 4.1) can be rejected. We will use level of significance $\alpha = 0.1$, as this is a first, exploratory analysis of our stated research questions based on a sample of limited size [Montgomery 2006]. For Binary scales, we test differences in proportions using Fisher's exact test [Yates Supplement 1, 1934], which is typically used to analyze contingency tables when samples are small. For testing time differences, we use the non-parametric Mann-Whitney test [Montgomery 2006].

Results in Table 4 clearly show that for Feasibility, Uncertainty, and Effort, the
impact of Slicing is statistically significant. The differences in Precision are, as expected, not significant; whereas, the differences in Recall are. This is important as in our context, as discussed earlier, high Recall values are crucial. Regarding the remaining four Correctness measurements, the impact of slicing is significant for two: it is significant when considering “I don’t know” answers to be incorrect instead of not including them in correctness assessments. The p-value is lower in the case where “Likely” answers are ignored (i.e., column 2 in Table 4, where uncertain answers are not accounted for), as these answers are more likely to be incorrect than “Definitely” answers (i.e., certain answers).

Table 4 Statistical test results.

<table>
<thead>
<tr>
<th>Hyp.</th>
<th>Variable</th>
<th>Test</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>H^110</td>
<td>Correctness; Yes Uncertain, Yes Unfeasible</td>
<td>Fisher exact test</td>
<td>0.0798</td>
</tr>
<tr>
<td>H^120</td>
<td>Correctness; Yes Uncertain, No Unfeasible</td>
<td>Fisher exact test</td>
<td>0.3244</td>
</tr>
<tr>
<td>H^130</td>
<td>Correctness; No Uncertain, Yes Unfeasible</td>
<td>Fisher exact test</td>
<td>0.0293</td>
</tr>
<tr>
<td>H^140</td>
<td>Correctness; No Uncertain, No Unfeasible</td>
<td>Fisher exact test</td>
<td>0.2760</td>
</tr>
<tr>
<td>H^150</td>
<td>Correctness; Recall</td>
<td>Fisher exact test</td>
<td>0.0031</td>
</tr>
<tr>
<td>H^160</td>
<td>Correctness; Precision</td>
<td>Fisher exact test</td>
<td>0.4759</td>
</tr>
<tr>
<td>H^20</td>
<td>Feasibility</td>
<td>Fisher exact test</td>
<td>0.0040</td>
</tr>
<tr>
<td>H^30</td>
<td>Uncertainty</td>
<td>Fisher exact test</td>
<td>0.0091</td>
</tr>
<tr>
<td>H^40</td>
<td>Effort</td>
<td>Mann-Whitney</td>
<td>&lt;&lt; 0.001</td>
</tr>
</tbody>
</table>

Table 5 Power analysis results.

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Actual proportion of control group</th>
<th>Observed effect size</th>
<th>Estimated minimum effect size (absolute improvement) for Power &gt; 0.80</th>
</tr>
</thead>
<tbody>
<tr>
<td>H^110</td>
<td>0.589</td>
<td>0.057</td>
<td>0.195</td>
</tr>
<tr>
<td>H^140</td>
<td>0.629</td>
<td>0.086</td>
<td>0.226</td>
</tr>
<tr>
<td>H^150</td>
<td>0.588</td>
<td>0.051</td>
<td>0.186</td>
</tr>
</tbody>
</table>

5.2.2 Power Analysis

5.2.2.1 Introduction

In many experimental fields of science, power analysis is performed prior to conducting an experiment as a way to plan an appropriate sample size. But in our context, like in most software engineering experiments, the number of subjects, and therefore the number of observations, is imposed by external constraints such as the limited availability of trained subjects and the high cost of conducting experiments. In this context, a retrospective power analysis, as suggested in [Thomas 1997], is useful for interpreting the non-significant results of statistical tests.

Some terminology needs to be introduced first before we explain our power analysis procedure. One key concept in power analysis is the effect size, which can be a measure of the strength of the relationship between two variables in a statistical population (actual effect size), or a sample-based estimate of that quantity (observed effect size) [Baguley 2006]. Specifically, the observed effect size may be significantly different from the actual effect size, especially on small samples, due to randomness effects introduced during the sampling procedure. In our case, effect size is measured
as the difference between the following two values (1) Correctness of the treatment group (i.e., group using slices) and (2) Correctness of the control group (i.e., group not using slices).

The **level of power** is the probability of rejecting the null hypothesis, given that the null is false and some alternative hypothesis is true [25].

### 5.2.2.2 Analysis Procedure

In this paper we applied the retrospective power analysis, as suggested in [Thomas 1997], to check whether our sample size is *large enough* to provide reliable results or we need a replica to achieve more confidence. Given the cost of traceability between requirements and design elements, as proposed in our methodology, we are only interested in *medium and large effects*; Cohen provides a classification of effect sizes for social sciences [Coehn 1992] -- effect sizes of 0.4 and 0.6 are classified as “medium” and “large”. The sample size of our experiment should be considered large enough if able to detect at least a medium effect at an *acceptable level of power* (typically 0.8).

Because sample size and power is obviously not an issue when we are able to reject our hypotheses, then we applied power analysis to the hypotheses that failed to be rejected. Specifically, for each dependent variable for which we did not observe any statistically significant result, we estimate the *minimum effect size* at which we obtain an acceptable level of power (80%). Therefore, the sample size of our experiment should be considered large enough for a specific dependent variable if, for a minimum effect size of 0.4, the estimated power is at least 0.8.

### 5.2.2.3 Results

Table 5 reports the results of power analysis for the dependent variables for which we did not observe any significant result. The table shows, in order, the tested hypothesis, the observed correctness proportion for the control group, the observed effect size on the sample, and the estimated, minimum effect size to obtain 80% power.

The table clearly shows that, for those variables, the observed effect sizes are much smaller than the estimated minimum effect size, hence explaining the lack of significance. But it also clearly shows that this minimum effect size is roughly around 0.2, much less than our 0.4 requirement. In other words, it means that for these variables we can detect relatively small effects with sufficient power.

For example, for the first dependent variable in Table 5, the observed effect size is 0.057 whereas 80% power is only achieved with an effect size of 0.195, that represents a difference between proportions. Such a difference in effect size, given a correctness proportion of 0.589 in the control group, would imply an actual correctness of 0.646 in the treatment group (i.e., 0.646 · 0.589 = 0.195).

### 5.2.2.4 Discussion

According to Cohen classification of effect sizes, our estimated minimum effect sizes (around 0.2) that would likely be detected (80% power) would be considered small. It is however not clear whether Cohen classification is meaningful in our application domain and the real question in our context is whether such effect sizes would justify the use of traceability. In any case, what we can conclude is that our experiment is able to detect, with a high level of confidence, effect sizes around 0.2. We are therefore confident that effect sizes above that threshold are not present and that the relatively small size of our sample is only a problem for effect sizes below 0.2, for the three dependent variables on which we did not obtain significant results. Only for such relatively small effect sizes, which are not of practical interest in our domain, our experiment would require replication and additional data points to draw reliable conclusions.

In conclusion, our retrospective power analysis suggests that the present experiment provides meaningful results in itself, without necessarily requiring replications and additional data points.
5.3 Additional Results

As discussed earlier, beyond slicing, there are two other factors that may have an effect on our dependent variables. To analyze the impact of the experiment steps and the type of defects on correctness, we will use the measure of correctness accounting for both unfeasible and uncertain answers (column 3 of Table 1). Again, this choice was made to maximize the number of observations.

5.3.1 Task

Fig.13 depicts the observations and regression lines – when making and not making use of slices – between (1) the time required to inspect the compliance of the design with respect to a single safety requirement and (2) the tasks of the experiments, i.e., the number requirements already inspected. As subjects work through the eight requirements, we might have expected learning or fatigue effects. In our experiment, as illustrated in Fig.11, as the subjects go through the inspection tasks, they become increasingly quicker at performing their task (negative slope). For example, when using slices, the average time is nearly divided by three from task 1 to task 8. The lower line in Fig.11 (dashed line) represents the case where slicing is used. Based on Standard Least Squares regression analysis, Task has a significant main effect (p-value << 0.001), and an insignificant interaction effect with slice (p-value=0.35) on required time. Such strong learning effects are fortunately handled by our experiment design. Interestingly this suggests that subjects are quickly improving the speed in checking compliance between requirements and SysML designs. The stable difference between the Slice and no-slice lines, suggests that slicing remains at least as useful when subjects are acquainted with the task as when they are starting.

There is no significant main (p-value=0.63) or interaction effect (p-value=0.22) of Task on Correctness, based on a Likelihood Ratio test when performing a logistic regression. This can be explained by the fact that subjects had ample time to perform their tasks, resulting in observable differences in time rather than correctness. It has been observed in past software engineering experiments that it is rather difficult to assess both a treatment effectiveness and its impact on the time needed to perform experimental tasks [Basili, et al. 1999]. Whether the effect, if any, is mostly visible on the time or effectiveness dimension is mostly driven by time constraints.

5.3.2 Type of Defect

From Fig. 14, we see that the use of slices increases correctness in similar proportions for both types of defects. Behavioural defects seem, however, to be overall easier to identify. This can be explained by the fact that behavioural defects are more di-
rectly related to the safety requirements being checked. The type of defect has a significant main effect (p-value=0.0649) and no significant interaction effect (p-value=0.45) with slice on correctness, based on a Likelihood Ratio test.

In Fig.15, the use of slices decreases the required time for both types of defects and the time it takes to find them is similar. The type of defect has no statistically significant main effect (p-value=0.64) or interaction effect (p-value=0.52) with slice on required time, based on Standard Least Squares regression.

5.4 Results Discussion

Overall, the results above show that the use of slicing, as presented in Section 3, had a significant positive effect on the capability of inspectors to identify inconsistencies between safety requirements and a SysML design. The effort to identify defects (inconsistencies) decreased by 27% on average with the use of slices. The number of subjects declining to answer decreased by 90% and the overall number of uncertain answers (“Likely” and “I don’t know”) decreased by 45%, thus suggesting an overall decrease in uncertainty in the presence of slices. Regarding the correctness of answers, the results are a bit more complex to discuss as correctness can be defined in different ways. In all cases, there is an increase in correctness but whether it is statistically significant depends on the definition used. For the perhaps more acceptable definition where “Likely” and “I don’t know” answers are accounted for, there is an increase in correctness from 50% to 63%. We believe that the reason why the effect is more visible for time than correctness is that the subjects had ample time to perform their tasks. This led them to using more time when no slice was available in order to achieve sufficient confidence about their answers.

Given that this is the first time a controlled experiment investigates the impact of traceability and model slicing on (safety) inspections of requirement-design compliance, these results are of high scientific and practical significance.

Traceability, a prerequisite for our slicing mechanism, is not without cost though. However, the overhead can be minimized if traces and their mappings are created while the designer is interpreting and designing the system, as the relevant information is readily available at that point. Whether the overhead of traceability is justified by the gains in quality and cost depends on many contextual factors. For safety-critical systems the overhead is most likely justified for two main reasons:

1) Industry standards require traceability between requirements and designs anyway;

2) The (safety-critical) system must be shown to be compliant with safety requirements, even when this entails additional cost.
6 THREATS TO VALIDITY

In the following we discuss the threats to validity related to our empirical procedure using the terminology and concepts defined by Wohlin et al. [Wohlin, et al. 2000]. This section provides a critical analysis of the results reported above; this will in turn enable the reader to compare them with results reported elsewhere and assess their applicability to her own context.

6.1 Conclusion Validity

Our main threat in this category is related to the fact that, as for many software engineering controlled experiments, we wished we would have more observations. Though we maximized their number through our experiment design, the statistical significance of our correctness results could be stronger with significantly more observations. Power analysis showed, however, that we can even be confident about our negative results for relatively small effect sizes.

6.2 Internal Validity

We have tried to balance all foreseeable factors that could trigger confounding effects with our independent variable, i.e., the use of slicing. The ability to control for such effects is the main strength of experiments such as the one we report here. In any experiment, the time to perform the tasks can be tightly constrained or be amply sufficient. In the latter case, one can then observe wide differences in effort across subjects, but not necessarily in terms of effectiveness [Basili, et al. 1999]. In the former case, usually subjects use up all the time available but then differences may be more visible in terms of effectiveness. Our experiment rather follows the first pattern and the effect of slicing on compliance decisions might be stronger with tighter time constraints.

The quality of the slices generated is ultimately related to the completeness and accuracy of the traceability information available. Three authors extensively reviewed the traceability links used in the experiment. The reported results should thus be interpreted as what one can expect with high-quality traceability information. In the future, we plan to investigate interaction effects between the quality of the traceability information and its presence, on the effectiveness of the inspection process.

6.3 Construct Validity

Measuring the effort involved in inspections was not particularly difficult but measuring compliance decision correctness could be performed in four main alternative ways. Because all of them can be considered valid, we performed the analysis using all four alternatives.

There are two social threats to construct validity that requires a short discussion. To minimize the impact of “hypothesis guessing” on results, we did not mention the experiment hypotheses during the training and pilot study. We believe that “evaluation apprehension” does not apply to the task of our experiment as students are not grades and, in any case, in realistic industrial environments inspectors are under pressure to detect safety issues.

6.4 External Validity

External validity is usually an issue in controlled experiments. Due to time constraints, tasks and artifacts tend to be small. This is certainly the case in our experiment as well, but our safety requirements and models are by no means trivial, as illustrated by the correctness rates of the subjects’ decisions. Further, if we compare the slices of our experiment artifacts with those of an earlier industrial case study [Nejati, et al. 2012], there are no significant differences in terms of size. In fact, our industrial case study involved the extraction of 34 block definition diagram slices and 31 activity diagram slices for a total of 30 (safety-relevant) requirements. On aver-
age, a block diagram slice included nine elements, an element being a block, a block operation, a block attribute, or an inter-block association. For activity diagrams, the average number of elements in a slice was 26, an element being an activity node or an activity transition. These sizes are respectively comparable to the size of block diagrams slices (six elements on average) and the size of activity diagram slices (21 elements on average) in our experiment. We note that our industrial case study did not involve the construction of state machines; therefore, no similar comparison can be made for state machine slices. But given the close relationship between activity diagrams and state machines in SysML, we anticipate the size of state machine slices in our controlled experiment to be comparable to those in real systems. Furthermore, we would expect the impact of using slices to increase for larger models and systems as inspecting complete models would then become increasingly error-prone and time consuming.

There are two additional threats to validity worth discussing. These are the Hawthorne effect and the placebo effect. The Hawthorne effect [Adair 1984] concerns the possibility that subjects may alter their performance due to being aware of being studied. In our experiment, subjects were carefully trained about the procedures to follow. The process they followed and the material they used were carefully controlled. It is difficult to imagine what they could have done differently because of being observed. The placebo effect [Rosenthal 1966] concerns the possibility that subjects may produce results that the researchers want without them realizing it. We avoided this threat by not exposing the subjects to the experimenters’ expectations.

7 CONCLUSIONS

This paper is concerned with inspections of compliance between (functional) safety requirements and system designs. This activity is a key component, required by many safety standards, of safety certification. We propose a methodology, supported by a tool, to model traceability between textual requirements and design models, and use such traces to extract slices (model fragments) containing only information that can be relevant to each requirement. We propose solutions specifically targeted at the System Modeling Language (SysML), an extension of UML, as this is becoming the de facto standard in system engineering [Schafer and Wehrheim 2007]. Since safety is a system issue, not just a software issue, this was a natural choice to ease the adoption of our proposal in practice.

The main contribution of this paper is the design and reporting of a controlled experiment aiming at assessing the impact of our methodology on inspectors' effectiveness and effort in inspecting requirements-design compliance. The experiment was designed to minimize threats to validity by maximizing the number of observations (conclusion validity) and balancing possible confounding factors (internal validity). Regarding external validity, our subjects were fully trained graduate students, many of them with industry experience, and our artifacts were composed of non-trivial SysML requirements and models. As for construct validity, the main issue was related to how the correctness of compliance decisions was measured and we decided to analyze the data using all plausible options.

Our experiment results clearly show that the proposed methodology for traceability and slicing brings practically significant advantages in terms of the correctness of inspections' outcomes, in terms of compliance decisions, and their effort. Since this is the first time a controlled experiment investigates the impact of traceability and model slicing on (safety) inspections of requirement-design compliance, these results are of high scientific and practical significance to industry sectors dealing with safety-critical systems.

We need to emphasize that model slicing is intended to be complementary to, rather than a substitute for, the complete design during inspections. The idea here is that the inspectors can always start with a slice first. Since the slices in our approach
are guaranteed to be sound, they do not mislead the inspectors into making incorrect decisions. However, the slices may turn out to be inconclusive due to incompleteness, in which case the inspectors will need to resort to the full design. What our experiment in this paper shows is that in the tradeoff between inconclusiveness versus the cognitive overload that the full design can cause, the latter (cognitive overload) is more likely to cause design issues to be overlooked, thus making model slicing advantageous.

Clearly, establishing the traceability links required by our model slicing approach involves an additional cost overhead. So future work will have to compare that overhead to the gains reported in this experiment. The extent of the overhead is however dependent on many contextual factors. In the context of the safety-critical systems we work with, standards require traceability between requirements and design anyways, and the decisions to be made are more about how to implement traceability, not whether to have it.

Being the first study of its kind, the experiment reported in this article is based on a small population size. For a small population, there is always a tradeoff between the number of levels of an independent variable (e.g. type of defects) and the number of observations per the variable. On the one hand, a low number of levels would yield results that are not easily generalizable, and on the other hand, a low number of observations would lead to difficulties in statistical interpretation of the results. In this article, given the population size, we focused on only four defect types, so that we could strike the right balance between generalizability and meaningful statistical analysis. Further research must be undertaken to analyze all defect types and eventually diverse instances of the same type. We provide all the material required to enable replications.

We further note that the defects we seeded into the models as part of our study, while based on issues observed in real settings, were not replayed from defects identified over earlier versions of the same models. This is because the PCS system used in our study is an exemplar and our models have not undergone independent inspections by expert safety inspectors. In the future, we plan to repeat our experiment using an industrial system and seed defects that have been actually detected by experts in the previous versions of the diagrams.

Another aspect of our future work has to do with analyzing the usefulness of slicing in a situation where the inspectors record the details of their reasoning about what is consistent and what is not. In our experiment, for time reasons, the subjects were not asked to delineate over the design the specific inconsistencies they found, nor where they asked to provide the rationale as to why they found a particular design consistent. It would be interesting to investigate to what extent the findings of our experiment carry over when the subjects need to provide finer-grained information about consistent and inconsistent designs.

Lastly, as we stated earlier, the quality of the slices generated is ultimately related to the completeness and accuracy of the traceability information available. One of the assumptions of the current experiment is the presence of high-quality traceability information. In the future we plan to investigate whether there is an interaction effect, between the quality of the traceability information and its presence, to the effectiveness of the inspection process.

8 ACKNOWLEDGMENT

Funding for the research was provided by DNV under the ModelME! project, by the Fonds National de la Recherche, Luxembourg (FNR/P10/03), and by the Research Council of Norway (Project No. 203461/030). We are grateful to Giovanni Cantone who taught modeling to the students involved in the experiment and Tonje Klykken who developed parts of the experiment material.
9 REFERENCES

J. Cleland-Huang, R. Settimi, O. BenKhadra, E. Berezhanskaya, and S. Christina. Year. Goal-centric traceability for managing non-functional requirements. ACM.


OMG. 2009. UML 2.2 Superstructure Specification (formal/2009-02-04).


Appendices

Appendix A

This appendix presents our traceability information model (Fig. 16). This model specifies the well-formedness criteria for the traceability links underlying our methodology in [Nejati, et al. 2012]. For a complete discussion on the information model, see [Nejati, et al. 2012].

Fig. 16 Traceability information model specifying the traceability links underlying our methodology in [Nejati, et al. 2012].
Appendix B

In this appendix, we present our algorithm for generating design slices relevant to a given safety requirement. For a thorough discussion of the algorithm, consult [Nejati, et al. 2012].

Algorithm. GENERATESLICE

Input: A block-level safety requirement, r.
A set of SysML design diagrams conforming to the traceability information model in Appendix 1.

Output: A block diagram slice related to r, Block.Slice_r.
An activity diagram slice related to r, Act.Slice_r.
A state machine slice related to r, SM.Slice_r.

/* Initialization. */

/*The trace and allocate links, the Mapping elements, and the relations between activity nodes and edges, state machine states and transitions, and block operations and states are shown in Appendix 1.*/

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 SM_r be the set of state machines related to the blocks in B_r via allocate links.
4. Let Block.Elem_r be the set of block states and operations related to r via Mapping elements.
5. Let Act.Elem_r be the set of activity nodes and edges related to the elements in Block.Elem_r.
6. Let SM.Elem_r be the set of state machine states and transitions related to the elements in Block.Elem_r.
7. Let Block.Slice_r, Act.Slice_r and SM.Slice_r be ∅

/* Step 1. Find elements temporally related to r (Design.Elem_r). */
9. for any block b ∈ B_r and any element e ∈ Design.Elem_r do
10. if operation op of b triggers (or is triggered by) e then
11. Design.Elem_r = Design.Elem_r ∪ {op}
12. for any activity partition a ∈ Act_r and any element e ∈ Design.Elem_r do
13. if activity node n of a triggers (or is triggered by) e then
14. Design.Elem_r = Design.Elem_r ∪ {n}
15. for any state machine sm ∈ SM_r and any element e ∈ Design.Elem_r do
16. if state s of sm triggers (or is triggered by) e then
17. Design.Elem_r = Design.Elem_r ∪ {s}

/* Step 2. Extract block diagram slices (Block.Slice_r). */
18. for every block b ∈ B_r, do
19. Let bOp and bAttr be the sets of operations and attributes of b, respectively.
20. bOp' = bOp ∩ Design.Elem_r
21. bAttr' = bAttr ∩ Design.Elem_r
22. Let bOp' and bAttr' be the new sets of operations and attributes of b, respectively.
23. Block.Slice_r = Block.Slice_r ∪ {b}
24. Block.Slice_r = Block.Slice_r ∪ {rel | rel is a block relation between blocks in B_r}
25. for every activity partition \( a \in \text{Act}_r \) do
26.   Let \( a\text{Nodes} \) and \( a\text{Edges} \) be the sets of nodes and edges of \( a \), respectively.
27.   Let \( \text{init} \) be the initial node of \( a \).
28.   /* Remove every node in \( a\text{Nodes} \) except for those in \( \text{Design}_r \), and the ending points of the activity edges in \( \text{Design}_r \). */
29.   \( a\text{Nodes}' = a\text{Nodes} \cap (\text{Design}_r \cup \{ \text{the ending points of the activity edges in } \text{Design}_r \}) \)
30.   /* Remove every edge in \( a\text{Edges} \) except for those whose ending points are in \( a\text{Nodes}' \). */
31.   \( a\text{Edges}' = \{ e \in a\text{Edges} \mid \text{such that both ending points of } e \text{ are in } a\text{Nodes}' \} \)
32.   Let \( a\text{Nodes}' \) and \( a\text{Edges}' \) be the new sets of nodes and edges of \( a \), respectively.
33.   /* Add stuttering edges. */
34.   for every pair \( n, n' \in a\text{Nodes}' \) do
35.     if \( n' \) is reachable from \( n \) through edges none of which are in \( a\text{Edges}' \) then
36.       add a stuttering edge from \( n \) to \( n' \)
37.   /* Pick a new initial node. */
38.   for every node \( n \) in activity partition \( a \) do
39.     if there is a path from \( \text{init} \) to \( n \) that does not go through any node in \( a\text{Nodes}' \) then
40.       mark \( n \) as a new initial node of \( a \).
41.   \( \text{Act}_r = \text{Act}_r \cup \{ a \} \)

/* Step 4. Extract state machine diagram slices (SM.Slice_r). */
42. for every state machine \( sm \in \text{SM}_r \) do
43.   Let \( sm\text{States} \) and \( sm\text{Trans} \) be the sets of states and transitions of \( sm \), respectively.
44.   Let \( \text{init} \) be the initial state of \( sm \).
45.   /* Remove every state in \( sm\text{States} \) except for those in \( \text{Design}_r \), and the ending points of the transitions in \( \text{Design}_r \). */
46.   \( sm\text{States}' = sm\text{States} \cap (\text{Design}_r \cup \{ \text{the ending points of the transitions in } \text{Design}_r \}) \)
47.   /* Remove every transition in \( sm\text{Trans} \) except for those whose ending points are in \( sm\text{States}' \). */
48.   \( sm\text{Trans}' = \{ t \in sm\text{Trans} \mid \text{such that both ending points of } t \text{ are in } sm\text{States}' \} \)
49.   Let \( sm\text{States}' \) and \( sm\text{Trans}' \) be the new sets of states and transitions of \( sm \), respectively.
50.   /* Add stuttering transitions. */
51.   for every pair \( s, s' \in sm\text{States}' \) do
52.     if \( s' \) is reachable from \( s \) in \( sm \) through transitions none of which are in \( sm\text{Trans}' \) then
53.       add a stuttering transition from \( s \) to \( s' \)
54.   /* Pick a new initial state. */
55.   for every state \( s \) in state machine \( sm \) do
56.     if there is a path from \( \text{init} \) to \( s \) that does not go through any state in \( sm\text{States}' \) then
57.       mark \( s \) as a new initial state of \( sm \).
58.   \( \text{SM}_r = \text{SM}_r \cup \{ sm \} \)
Appendix C

In this appendix, we present the original state machine diagrams (Fig. 17) related to the blocks in Fig. 1 together with the state machine diagram slice (Fig. 18) related to $R1$ in Fig. 1.

Fig. 17 The state machine diagrams related to the blocks in Fig. 1. The states and transitions relevant to $R1$ are highlighted with the thick lines.

Fig. 18 The state machine slice related to $R1$ in Fig. 1.

Appendix D  Safety requirements

The ten requirements used in our work are provided in Table 6. The first eight requirements were used in the experiment and last two were used in the pilot study. The PCS has four system-level safety requirements described below:

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

- **Avoid machine collisions.** There should not be any collision between PCS devices.
- **Avoid falling metal blanks/plates.** The blanks must not fall on the ground.
- **Avoid piling or overlapping blanks/plates.** Blanks should not be piled on each other, overlapping, or placed too close for being distinguished by PCS photoelectric cells.

Each of the requirements in Table 6 is related to one of the above system-level requirements. Specifically, the first requirement is related to the **avoid machine collisions** constraint. This is because robot and press may collide if the press moves to the middle position while the robot arm has remained extended. The second requirement is related to the **avoid falling metal blanks** constraint. The third requirement is also related to the **avoid falling metal blanks** constraint. In other words, if the crane does not pick up the existing blank (or plate) on the deposit belt and the deposit belt starts moving, the blank will fall down. The forth requirement is related to the **avoid piling or overlapping blanks** constraint. More precisely, the feedbelt needs to be informed when the table becomes empty, so that it is not going to put a new blank on a non-empty table. The fifth requirement is related to the **restrict machine mobility** constraint. The sixth requirement is related to the **avoid piling or overlapping blanks** constraint. This is because the robot needs to be informed when the blank reaches the end of the deposit belt, so that it does not put too many blanks on the deposit belt (i.e., the robot puts the blanks on the deposit belt in such a way that a minimum distance is maintained between the blanks). The seventh requirement is related to the **restrict machine mobility** constraint. The eighth requirement is related to the **avoid piling or overlapping blanks** constraint. The ninth requirement is related to the **avoid machine collisions** constraint, and the tenth requirement is related to the **restrict machine mobility** constraint. Finally, the set of all the slices related to the requirements in Table 6 is available at [http://people.svv.lu/nejati/ExperimentMaterial/](http://people.svv.lu/nejati/ExperimentMaterial/).

### Table 6 Requirements used in the controlled experiment and the pilot study.

<table>
<thead>
<tr>
<th>Req. #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>After robot loads the press, its upper arm (arm1) must be retracted or the press must move away from the middle position.</td>
</tr>
<tr>
<td>2</td>
<td>The magnet of arm1 of the robot may only be deactivated if it is inside the press.</td>
</tr>
<tr>
<td>3</td>
<td>The deposit belt may only be started (i.e., turn on its motor) after the crane has picked up a blank from the deposit belt.</td>
</tr>
<tr>
<td>4</td>
<td>When no blank is on the table, the table informs the feedbelt to put another blank on it.</td>
</tr>
<tr>
<td>5</td>
<td>Both robot arms must not be retracted or extended more than necessary. That is, a retracted robot arm must be extended before retracting again. Similarly, an extended robot arm must be retracted before extending again.</td>
</tr>
<tr>
<td>6</td>
<td>When a plate has arrived at the end of the deposit belt, the deposit belt informs the robot. Subsequently the robot (using arm2) puts a new plate on the deposit belt.</td>
</tr>
<tr>
<td>7</td>
<td>If the crane is positioned above the deposit belt, it may only move towards the container.</td>
</tr>
<tr>
<td>8</td>
<td>A new blank may only be put on the feed belt if the former one has arrived on the table.</td>
</tr>
<tr>
<td>9</td>
<td>The press may only move when no robot arm is positioned inside it.</td>
</tr>
<tr>
<td>10</td>
<td>The table must not be rotated counterclockwise if it is in unloaded position (position required for transferring blanks to the robot).</td>
</tr>
</tbody>
</table>
Appendix E  Seeded defects with examples

Incorrect navigation of an association (S0) Fig. 19 shows a fragment of the slice related to Req. #3 in Table 7 with a seeded defect of type S0. In the correct design (see http://people.svv.lu/nejati/ExperimentMaterial/req16/trace-report-req16.pdf), the navigation of the association between classes DepositBelt and UnidirectionalMotor is from DepositBelt to UnidirectionalMotor. The block slice in Fig. 19 violates Req. #3 because as described in the text of the requirement, the deposit belt has a motor part and not the other way around.

Incorrect multiplicity of an association (S1) Fig. 20 shows a fragment of the slice related to Req #5 in Table 7 with a seeded defect of type S1. In the correct design (see http://people.svv.lu/nejati/ExperimentMaterial/req2/trace-report-req2.pdf), the multiplicity of class Arm side of the association between classes Robot and Arm is exactly “2” instead of “0..3” as in Fig. 20. The block slice in Fig. 20 violates Req #5 because as described in the text of the requirement, the robot has exactly two arms.

Wrong temporal ordering of operations of an activity diagram (B0) Fig. 21 shows a fragment of the slice related to Req #8 in Table 7 with a seeded defect of type B0. In the correct design (see http://people.svv.lu/nejati/ExperimentMaterial/req17/trace-report-req17.pdf), the add_blank action is followed by the feed_table action instead of another add_blank action. Obviously, the activity slice in Fig. 21 violates Req #8 because it describes a situation where two blanks are put on the feedbelt consecutively before the first one arrives the table.

Incorrect effect on transitions of a state machine diagram (B1) Fig. 22 shows a fragment of the slice related to Req #2 in Table 7 with a seeded defect of type B1. In the correct design (see http://people.svv.lu/nejati/ExperimentMaterial/req12/trace-req12.pdf), the effect (or output) on the transition from state Arm extend to Press to state Arm retract from Press is arml.retract() instead of arml.extend(). In addition, the exit action of state Arm extend to Press is arml.drop() instead of arml.extend(), and further, the exit action of state Arm retract from Press is empty instead of arml.drop(). Obviously, the state machine slice in Fig. 22 violates Req #2 because arml.magnet.demagnetize() is called after three calls to arml.extend(). This implies that the magnet of arml is deactivated when it is not located inside the press, violating Req #2.
Requirement:

R3 = The deposit belt may only be started (i.e., turn on its motor) after the crane has picked up a blank from the deposit belt.

Traceability:

\[ \text{depositBelt, conveying\_motor, turn\_on} \] causes "the deposit belt is started"

After crane pick from belt(), "the crane has picked up a blank from the deposit belt"

Diagram Slices:

Block Slices:

Fig. 19 Example of Defect S0

---

Requirement:

R5 = Both robot arms must not be retracted or extended more than necessary. That is, A retracted robot arm must be extended before retracting again. Similarly, an extended robot arm must be retracted before extending again.

Traceability:

\[ \text{arm\_retract, arm\_extend} \] causes "the upper robot arm to be retracted"

\[ \text{arm\_extend, arm\_retract} \] causes "the upper robot arm to be extended"

Diagram Slices:

Block Slices:

Fig. 20 Example of Defect S1
Requirement:

R8 = A new blank may only be put on the feed belt if the former one has arrived on the table.

Traceability:

After feedbelt.feed_table(), "a blank is put on the table"

feedbelt.add_blank(), causes "a new blank to be put on the feedbelt"

Diagram Slices:

Block Slices:

Activity Slices:

Fig. 21 Example of Defect B0
Fig. 22 Example of Defect B1
Appendix F  Answer sheet

The answer sheet used in the experiment is shown in Fig. 23.

ID Requirement: 18
ID Subject: 
Step: 
Initial time (hh:mm.seconds): 

Requirement Description: When a plate has arrived at the end of the deposit belt, the deposit belt informs the robot. Subsequently the robot (using arm2) puts a new plate on the deposit belt.

How is the design in respect to the requirement?  
0: I don’t know. 1: Definitely consistent. 2: Likely consistent. 3: Likely inconsistent. 4: Definitely inconsistent.

Location(s) (pages and/or figure number):

If any, please check the type (only one) of defect that is present:

<table>
<thead>
<tr>
<th>Block Definition Diagram</th>
<th>Activity Diagram</th>
<th>State Machine Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Missing Block</td>
<td>Missing Activity Node</td>
<td>Missing Region</td>
</tr>
<tr>
<td>Missing Association</td>
<td>Missing Action Node</td>
<td>Missing State</td>
</tr>
<tr>
<td>Missing Generalization</td>
<td>Missing Fork Node</td>
<td>Missing Transition</td>
</tr>
<tr>
<td>Missing Dependency</td>
<td>Missing Join Node</td>
<td>Incorrect State</td>
</tr>
<tr>
<td>Incorrect Block</td>
<td>Missing Decision Node</td>
<td>Missing Entry Behavior</td>
</tr>
<tr>
<td>Missing Part Property</td>
<td>Missing Merge Node</td>
<td>Missing Exit Behavior</td>
</tr>
<tr>
<td>Missing Value Property</td>
<td>Missing Data Store Node</td>
<td>Missing Do Activity</td>
</tr>
<tr>
<td>Missing Port and Flow</td>
<td>Missing Activity Edge</td>
<td>Incorrect state type</td>
</tr>
<tr>
<td>Missing Operation</td>
<td>Missing Data Flow</td>
<td>Incorrect Entry</td>
</tr>
<tr>
<td>Incorrect Property</td>
<td>Missing Control Flow</td>
<td>Incorrect Exit</td>
</tr>
<tr>
<td>Incorrect Port and Flow</td>
<td>Missing Activity Partition</td>
<td>Incorrect Do Activity</td>
</tr>
<tr>
<td>Incorrect Operation</td>
<td>Missing Interruptible Region</td>
<td>Incorrect Transition</td>
</tr>
<tr>
<td>Incorrect Block</td>
<td>Missing Precondition</td>
<td>Missing Trigger</td>
</tr>
<tr>
<td>Missing Part Property</td>
<td>Missing Postcondition</td>
<td>Missing Guard condition</td>
</tr>
<tr>
<td>Missing Value Property</td>
<td>Incorrect Activity Node</td>
<td>Missing Effect</td>
</tr>
<tr>
<td>Missing Port and Flow</td>
<td>Incorrect Activity Node Parameter</td>
<td>Incorrect Trigger</td>
</tr>
<tr>
<td>Missing Operation</td>
<td>Incorrect Action Node</td>
<td>Incorrect Guard condition</td>
</tr>
<tr>
<td>Incorrect Property</td>
<td>Incorrect Action Type</td>
<td>Incorrect Effect</td>
</tr>
<tr>
<td>Incorrect Port and Flow</td>
<td>Missing InputPin</td>
<td></td>
</tr>
<tr>
<td>Incorrect Operation</td>
<td>Incorrect InputPin</td>
<td></td>
</tr>
<tr>
<td>Incorrect Association</td>
<td>Wrong temporal ordering of operation</td>
<td></td>
</tr>
<tr>
<td>Incorrect Multiplicity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incorrect Navigation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incorrect Generalization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incorrect Dependency</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Other:

Final time (hh:mm.seconds):

Fig. 23 The Answer Sheet