Automated Change Impact Analysis between SysML Models of Requirements and Design

Shiva Nejati, Mehrdad Sabetzadeh, Chetan Arora, Lionel C. Briand
SnT / University of Luxembourg, Luxembourg
{firstname.lastname}@uni.lu

Felix Mandoux
Delphi Automotive, Luxembourg
felix.mandoux@delphi.com

ABSTRACT

An important activity in systems engineering is analyzing how a change in requirements will impact the design of a system. Performing this analysis manually is expensive, particularly for complex systems. In this paper, we propose an approach to automatically identify the impact of requirements changes on system design, when the requirements and design elements are expressed using models. We ground our approach on the Systems Modeling Language (SysML) due to SysML's increasing use in industrial applications.

Our approach has two steps: For a given change, we first apply a static slicing algorithm to extract an estimated set of impacted model elements. Next, we rank the elements of the resulting set according to a quantitative measure designed to predict how likely it is for each element to be impacted. The measure is computed using Natural Language Processing (NLP) applied to the textual content of the elements. Engineers can then inspect the ranked list of elements and identify those that are actually impacted. We evaluate our approach on an industrial case study with 16 real-world requirements changes. Our results suggest that, using our approach, engineers need to inspect on average only 4.8% of the entire design in order to identify the actually-impacted elements. We further show that our results consistently improve when our analysis takes into account both structural and behavioral diagrams rather than only structural ones, and the natural-language content of the diagrams in addition to only their structural and behavioral content.

CCS Concepts

•**Software and its engineering → Software evolution;**

Keywords

Change Impact Analysis, SysML, Traceability Information Model, Model Slicing, Natural Language Processing.

1. INTRODUCTION

Change impact analysis is an important activity in software maintenance and evolution, both for properly implementing a set of requested changes, and also for estimating the risks and costs associated with the change implementation [1, 2]. In addition to being a general best practice, change impact analysis is often mandatory for safety-critical applications and meeting the compliance provisions of safety standards such as IEC 61508 [3] and ISO 26262 [4].

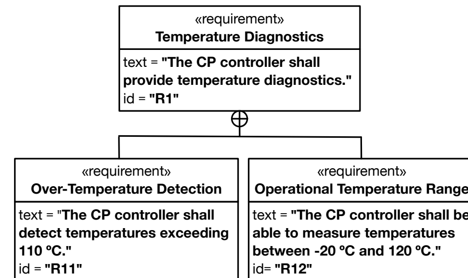

Figure 1: Requirements diagram fragment for CP.

In this paper, we concern ourselves with analyzing the impact of requirements changes on system design. Changes in requirements may occur due to a variety of reasons, including, for example, evolving user needs and budget constraints. Irrespective of the cause, it is important to be able to assess how a requirements change affects the design. Doing so requires engineers to identify, for each requirements change, the system blocks and behaviors that will be impacted. If done manually, this task can be extremely laborious for complex systems, thus making it important to support the task through automation.

Motivating Example. We motivate our work using a cam phaser (CP) system, developed by Delphi Automotive. This system, which includes mechanical, electronic and software components, enables adjusting the timing of cam lobes with respect to that of the crank shaft in an engine, while the engine is running. CP is safety-critical and subject to ISO 26262 – a functional safety standard for automobiles. To protect confidentiality and facilitate illustration, we have, in the description that follows, altered some of CP's details without affecting CP's core architecture and behavior.

The system requirements and the design of CP are expressed using the Systems Modeling Language (SysML) [5]. Figure 1 shows a small requirements diagram adapted from CP's original SysML models. The requirement on the top, Temperature Diagnostics (R1), is decomposed into two subrequirements: Over-Temperature Detection (R11) and Operational Temperature Range (R12). The over-temperature threshold specified by R11 and the operational temperature range specified by R12 depend on the specific devices that interact with CP and may vary from one engine configura-

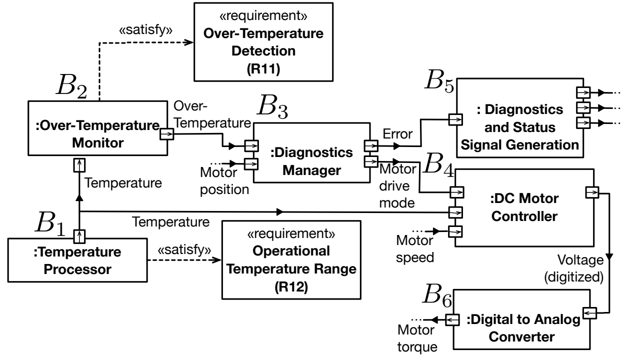

Figure 2: Fragment of CP's block diagram.

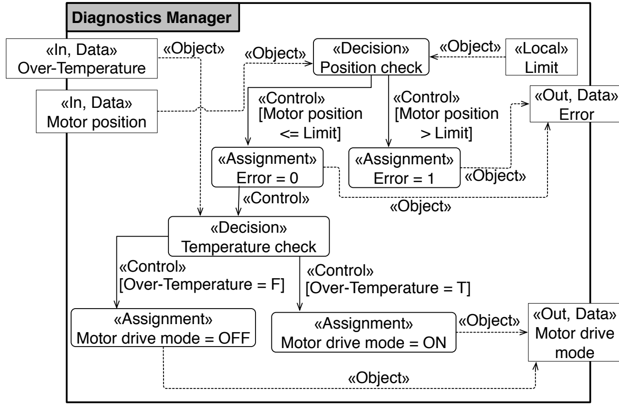

Figure 3: (Simplified) activity diagram for the Diagnostics Manager block (B_3) of Figure 2.

tion to another. Hence, it is common for systems engineers to receive change requests regarding these requirements. Examples of change requests coming from customers (typically, car manufacturers) and concerning these requirements are: (1) Ch-R11: *Over-temperature threshold shall change from 110 °C to 147 °C*, and (2) Ch-R12: *Temperature range shall be extended to -40 – 150 °C from -20 – 120 °C*.

Figures 2 and 3 present parts of CP's design: Figure 2 shows a fragment of CP's architecture expressed as a SysML internal block diagram. In this diagram, there are two traceability links to requirements, one from the Over-Temperature Monitor block (labeled B_2) to requirement R11, and the other from the Temperature Processor block (B_1) to requirement R12. Figure 3 shows an activity diagram describing the behavior of the Diagnostics Manager block (B_3). For succinctness, we take the term "block" to represent *instances* of SysML block types. This choice does not cause ambiguity in our presentation, as our motivating example does not have multiple instances of the same block type.

A simple intuition that systems engineers apply for scoping the impact of requirements changes is to follow the flow of data between the design blocks, starting from the blocks that are directly traceable to the changed requirements. For example, for Ch-R11, one would start from block B_2, which is directly traced to R11, and mark as potentially-impacted any block that is reachable from B_2 via the inter-block connectors. Using this kind of reasoning, we obtain the following estimated impact sets for Ch-R11 and Ch-R12, respectively: $\{B_2, B_3, B_4, B_5, B_6\}$ and $\{B_1, B_2, B_3, B_4, B_5, B_6\}$.

Estimating the impact sets in the manner described above, i.e., by reachability analysis over the inter-block connectors, often yields too many false positives, i.e., too many blocks that are not actually impacted by the change under investigation. For example, we know from a manual inspection conducted by the systems engineers involved that the actual impact sets for Ch-R11 and Ch-R12 are $\{B_2\}$ and $\{B_1, B_4, B_6\}$, respectively. This means that B_3, B_4, B_5 and B_6 in the estimated impact set for Ch-R11, and B_2, B_3 and B_5 in the estimated impact set for Ch-R12 are false positives.

Some of these false positives can be pruned by considering the block behaviors. To illustrate, consider the activity diagram of Figure 3. By following the control and data flows in this diagram, we can infer that (1) the Motor position input may influence both the Error and Motor drive mode outputs, and (2) the Over-Temperature input may influence only the Motor drive mode output. We can therefore conclude that B_5 is unlikely to be impacted by Ch-R11 and Ch-R12, and thus remove B_5 from the estimated impact sets above.

Despite the analysis of block behaviors being helpful for pruning the estimated impact sets, such analysis alone does not adequately address imprecision, still leaving the engineers with a large number of false positives and hence a large amount of wasted inspection effort. To further improve precision, we recognize that there is a wealth of textual content in the models, e.g., the labels of blocks, ports and actions. This raises the possibility that text analysis can be a useful aid for making change analysis more precise.

To this end, we use insights from our previous work on the propagation of change in natural-language content [6]. In particular, we have observed that, alongside the change description, one can further obtain cues from the engineers about how they expect a given change to propagate. For example, the engineers of CP could provide the following intuition about the impact of Ch-R12 on the design, *before* actually inspecting the design: "Temperature lookup tables and voltage converters need to be adjusted".

From the description of Ch-R12 and the intuition above given by the engineers, it is reasonable to expect that a block containing one or more of the keyphrases "temperature range", "temperature lookup table", and "voltage converter" (or similar phrases) should have a higher likelihood of being impacted by Ch-R12 than a block that contains none of these keyphrases. Indeed, the keyphrase "temperature lookup table" appears in the action nodes of the activity diagrams that describe the block behaviors of B_1 and B_4 (not shown), thus making B_1 and B_4 more likely to be impacted than other blocks, say B_2 and B_3, whose activity diagrams do not contain this keyphrase. In a similar vein, the keyphrase "voltage converter" mentioned by the engineers will increase the likelihood of impact on B_6 as compared to B_2 and B_3.

Contributions. We propose an automated approach for identifying the impact of requirement changes on system design. Our approach takes into account all the intuitions illustrated on the motivating example described above, utilizing the inter-block connectors, the block behaviors, and the textual content of the models for increasing the precision of change impact analysis. Our approach has two steps: For a given change, we first compute an *estimated impact set* by identifying the design elements that are reachable from the changed requirement. The basis for this step are the inter-block connectors and block behaviors. The main novelty of this step is in providing a rigorous adaptation of dependency graphs – commonly used in program slicing [7] – for reachability analysis over the activity diagrams that describe the

block behaviors. In the second step, we automatically rank the elements of the estimated impact set. The ranking is aimed at predicting how likely it is for each element in this set to be affected by the given change. The basis for the ranking is a quantitative measure computed using Natural Language Processing (NLP) [8]. Specifically, the measure reflects the similarity between the textual content of the elements in the estimated impact set and the keyphrases in the engineers' statement about the change. We provide guidelines for deciding about the cutoff point in the ranked list; this is the point beyond which the elements in the list would not be worthwhile inspecting because their likelihood of being impacted is low. The novelty of the second step of our approach is in applying NLP for change analysis between modeling artifacts (as opposed to textual artifacts).

While the ideas behind our work are general, we ground our approach on SysML. This choice is motivated primarily by two factors: First, SysML is representing a significant and increasing segment of the embedded software industry, particularly in safety-critical domains. Given the importance of change impact analysis for complying with safety standards, we believe that building on SysML is advantageous as a way to facilitate the integration of our approach into safety certification activities. Second, SysML provides a built-in mechanism, via requirements diagrams, for connecting design models to natural-language requirements. This allows us to capitalize as much as possible on the standard requirements-to-design trace link in SysML.

We implement our approach as a plugin for Enterprise Architect [9]. We report on an industrial case study conducted in collaboration with Delphi Automotive, which is an international supplier of vehicle technology. The case study includes 16 real-world requirements changes.

Our results indicate that the number of elements engineers need to inspect decreases as we combine different sources of information. In particular, on average, this number is 21.6% of the entire design (80 / 370 design elements) when we consider only inter-block connectors. This average reduces to 9.7% (36 / 370) when we consider both inter-block connectors and block behaviors. The average further reduces to 4.8% (18 / 370) when we also take into account the natural-language information in the models and the engineers' change statements. The precision of our approach when all the above three sources of information are used is 29.4%. That is, on average, 29.4% of a set consisting of 18 elements is actually impacted. Given that the approach narrows potentially-impacted elements to a small set (4.8% of the design), excluding false positives from the results can be done without substantial effort. Our analysis misses one impacted element for only one out of the total of 16 changes. The recall is 85% for that particular change and 100% for the other 15 changes, giving an average recall of 99%.

Structure. Section 2 describes our approach. Section 3 presents our empirical evaluation. Section 4 compares with related strands of work. Section 5 concludes the paper.

2. APPROACH

Figure 4 shows an overview of our change impact analysis approach. In this section, we first describe the modeling prerequisites for our approach. We then elaborate the steps of our approach, marked 1 and 2 in Figure 4.

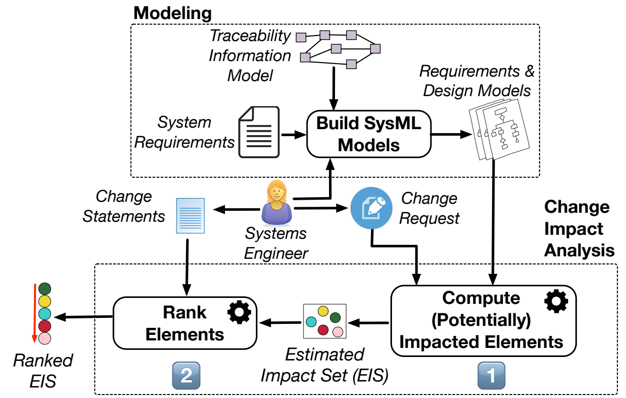

Figure 4: Approach overview.

2.1 Building SysML Models

Our approach concentrates on models built along three dimensions: (1) requirements, (2) structure (architecture), and (3) behavior. We use SysML requirements diagrams, illustrated in Figure 1, for expressing requirements. We model the system structure using SysML internal block diagrams, illustrated in Figure 2. Finally, we use SysML activity diagrams, illustrated in Figure 3, for capturing behaviors. Our focus on these three model types is in line with SysML's core modeling practices [10, 11] and further with the modeling choices made by Delphi Automotive.

Our change impact analysis approach is agnostic to the particular modeling methodology used to build the above model types, as long as the methodology provides the traceability information required by our approach. We characterize the required traceability information via the *traceability information model (TIM)* of Figure 5. TIMs are a common way of specifying how different development artifacts (and the elements thereof) should be traced to one another in order to support specific analytical tasks [12, 13, 14].

Our TIM is organized into three packages, representing the three modeling dimensions covered. This TIM is consistent with both the existing literature on systems engineering modeling [15], and also the SysML/UML metamodel [10, 11]. We note that providing the information envisaged by our TIM does not require significant additional manual work. Specifically, all the elements and associations in our TIM, except for the satisfy association from blocks to requirements, are implied by the natural process of model construction in SysML. As for the links between requirements and system blocks prescribed by our TIM, establishing these links, irrespective of whether our change impact analysis technique is used or not, is one of the most basic best practices in SysML and, in the case of safety-critical applications, an essential prerequisite for complying with safety standards [16, 17, 18].

The requirements package in Figure 5 defines the requirements types, namely, software and hardware. Due to lack of space, Figure 5 does not show all the possible relations between the requirements (e.g., decomposition and derivation). Requirements may be connected to blocks via the satisfy relation. A satisfy link between a block B and a requirement R indicates that the function implemented by B contributes to the satisfaction of R. Blocks belong to the structure package, and can be either software or hardware. Each block, irrespective of its type, contains a number of ports. Ports are connected via the connector relation.

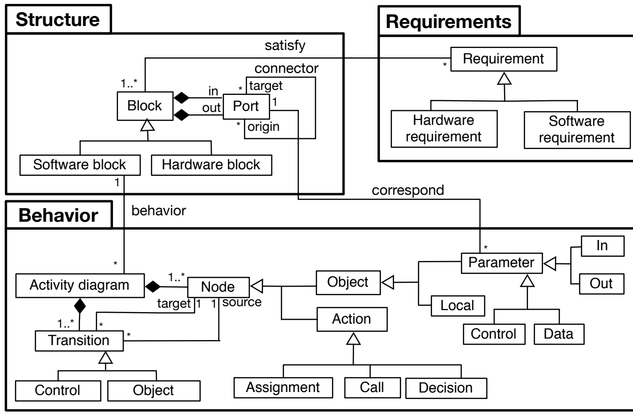

Figure 5: The traceability information required by our change impact analysis approach.

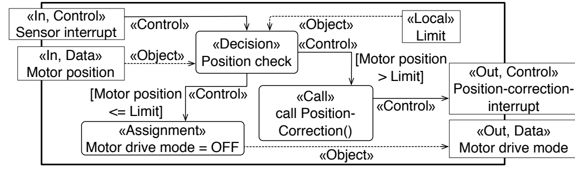

Figure 6: Example activity diagram with control input/output and a call action node.

Blocks can be associated with multiple behaviors (use cases). Each behavior is specified using one activity diagram. The behavior relation links blocks to their corresponding behaviors. Activity diagrams include nodes and transitions. Nodes can be either objects or actions; transitions may be of either the object or control kinds. Object transitions represent data flows (data dependencies), and control transitions represent control flows (control dependencies). Action nodes may be of one of the following three kinds: (1) assignment statements defined over parameters or local variables, (2) decision statements (if-statements) defined over parameters or local variables, and (3) call statements providing the behavior of function calls.

Object nodes can be designated as either parameter or local. Parameters represent the input/output variables of activity diagrams and may be of type control or data. Local nodes are used to store local variables. Control input/output variables and call action nodes are used for modeling the sending and receiving of events between blocks with concurrent behaviors. Finally, the correspond relation in Figure 5 indicates that each block port has some corresponding parameter in some activity diagram related to that block. Each activity diagram parameter has to correspond to one port of the block related to that activity diagram.

The diagrams in our motivating example of Section 1 conform to the TIM of Figure 5. Specifically, the internal block diagram of Figure 2 specifies the satisfy relations between the requirements of Figure 1 and blocks B_1 and B_2. The activity diagram of Figure 3 captures one possible behavior for block B_3. This activity diagram contains two input and two output parameters, all of type data, as well as a local variable, Limit. The input/output parameter nodes of the diagram correspond to the input and output ports of B_3.

There are six action nodes in the activity diagram of Figure 3: four assignments and two decisions. The call action node is illustrated in the activity diagram of Figure 6. In this figure, the control input Sensor interrupt models interrupt calls that can be periodic or aperiodic. The call action node call Position-Correction() produces a control output value, which is in turn used as a control input by another block.

Algorithm ComputeiImpact [using *only* inter-block structural relations].
Input: - A set R of requirements modified in response to a change request.
- Sets $\mathcal{B}$ (blocks), $\mathcal{P}$ (ports) and $\mathcal{AD}$ (activity diagrams), together with relations *satisfy*, *connect*, *in*, *out*, and *behavior*.

Output: - A set *EIS* of blocks, ports and activity diagrams impacted by R.

1. $iB = \bigcup_{r \in R} satisfy(r)$
2. $iP = \bigcup_{b \in iB} out(b)$
3. $iAD = \bigcup_{b \in iB} behavior(b)$
4. **do**
5. $\quad iP = iP \cup \bigcup_{p \in iP} connect(p)$
6. $\quad iB = iB \cup \bigcup_{p \in iP} in^{-1}(p)$
7. $\quad iAD = iAD \cup \bigcup_{b \in iB} behavior(b)$
8. $\quad iP = iP \cup \bigcup_{b \in iB} out(b)$
9. **until** $iB \cup iP \cup iAD$ *reaches a fixed point*
10. **return** $EIS = iB \cup iP \cup iAD$

Figure 7: Algorithm for computing an estimated impact set (*EIS*) using inter-block structural relations.

2.2 Computing Potentially Impacted Elements

In this section, we describe the first step of our change impact analysis approach (marked 1 in Figure 4). This step includes two algorithms: (1) computing reachability over inter-block structural relations, and (2) slicing activity diagrams based on intra-block behavioral relations.

Reachability over inter-block structural relations. We denote the sets of all requirements, blocks, ports and activity diagrams by $\mathcal{R}$, $\mathcal{B}$, $\mathcal{P}$, and $\mathcal{AD}$, respectively. Let $r \in \mathcal{R}$, $b \in \mathcal{B}$, and $p \in \mathcal{P}$. We write $satisfy(r) \subseteq \mathcal{B}$ to denote the set of blocks related to r by a satisfy relation. We denote the input ports of b by $in(b) \subseteq \mathcal{P}$, and its output ports by $out(b)$. We write $connect(p) \subseteq \mathcal{P}$ to indicate the set of ports related to p by connector links emanating from p. Finally, we denote by $behavior(b) \subseteq \mathcal{AD}$ the set of activity diagrams that specify the behaviors of b.

Figure 7 shows the algorithm for computing reachability over inter-block connectors (structural relations). The algorithm receives as input the set R of requirements modified in response to a change request. It then computes the initial sets iB of impacted blocks, iP of impacted ports, and iAD of impacted activity diagrams (lines 1–3). The initial set iB is obtained by following the satisfy links from the requirements in R. The set of output ports of the impacted blocks in iB are then stored in iP, and the activity diagrams related to the blocks in iB are stored in the initial set iAD.

After obtaining the initial sets, a transitive closure is computed over these sets (lines 4–9). In particular, by following the connector links originating from ports in iP, new input ports belonging to new blocks are identified. The new input ports are added to iP (line 5), and the new blocks are added to iB (line 6). We then identify the activity diagrams related by the behavior links to the newly added blocks, and add them to iAD (line 7). We further add the output ports of the newly added blocks to iP (line 9). Lines 5–8 are executed until we reach a fixed point. The result is an estimated impact set *EIS*. Note that for any activity diagram in *EIS*, we consider all the activity and object nodes in that diagram to be impacted.

Given the example diagrams in Figures 1–3, if we execute the algorithm of Figure 7 for change requests Ch-R11 and Ch-R12 (i.e., by setting the input set R to {R11} and {R12} respectively), the algorithm will respectively return $\{B_2, B_3, B_4, B_5, B_6\}$ and $\{B_1, B_2, B_3, B_4, B_5, B_6\}$ for iB. Further, the algorithm returns the set of all the ports

Algorithm ComputeiImpact [using *both* inter-block structural and intra-block behavioral relations].
Input: - A set R of requirements modified in response to a change request.
- Sets $\mathcal{B}$ (blocks), $\mathcal{P}$ (ports), and $\mathcal{AD}$ (activity diagrams) together with the relations *satisfy*, *connect*, *in*, *out*, and *behavior*.
Output: - A set *EIS* of blocks, ports and activity diagrams impacted by R.

```
1.  iB = ⋃_{r∈R} satisfy(r)
2.  iP = ⋃_{b∈iB} out(b)
3.  iAD = ⋃_{b∈iB} behavior(b)
4.  do
5.    iP = iP ∪ ⋃_{p∈iP} connect(p)
6.    for b ∈ ⋃_{p∈iP} in^{-1}(p) do
7.      iB = iB ∪ {b}
8.      for ad ∈ behavior(b) do
9.        iAD′, iOut = ForwardSlice(ad, correspond^{-1}(iP ∩ in(b)))
10.       iAD = iAD ∪ iAD′
11.       iP = iP ∪ correspond(iOut)
12. until iB ∪ iP ∪ iAD reaches a fixed point
13. return EIS = iB ∪ iP ∪ iAD
```

Figure 8: Algorithm for computing *EIS* using both structural and behavioral relations.

of the blocks in iB for iP and the activity and object nodes in Figure 3 for iAD.

Slicing activity diagrams based on intra-block behavioral relations. Our slicing algorithm operates on data and control flow dependencies among object and action nodes of activity diagrams. We first provide a formalization of activity diagrams' syntax and specify the notions of data and control dependency in our formalization. We then present our slicing technique for activity diagrams. An activity diagram ad is a tuple $\langle AN, V, ON, G, TC, TO\rangle$ where:

- AN is a set of action nodes, partitioned into three subsets AN^a, AN^c, and AN^d representing assignment, call, and decision action nodes, respectively.

- V is a set of variable names, partitioned into V^{io} and V^l indicating input/output and local variable names, respectively.

- ON is a set of object nodes, partitioned into two subsets ON^p and ON^l indicating parameter and local object nodes, respectively. The set ON^p of parameter nodes is partitioned into $ON^{p,c}$ and $ON^{p,d}$ indicating call and data object nodes, respectively. The set ON^p is also partitioned into $ON^{p,i}$ and $ON^{p,o}$ denoting input and output object nodes, respectively. We thus have: $ON^p = ON^{p,i} \cup ON^{p,o} = ON^{p,c} \cup ON^{p,d}$. Each object node in ON^l (resp. ON^p) is labeled with a variable name in V^l (resp. V^{io}).

- G is a set of Boolean expressions over the variables in V. These expressions capture transition guards.

- TC is a set of control transitions defined as follows: $TC \subseteq ((ON^{p,c} \cap ON^{p,i}) \times AN) \cup (AN^c \times (ON^{p,c} \cap ON^{p,o})) \cup ((AN^a \cup AN^c) \times AN) \cup (AN^d \times G \times AN)$. That is, control transitions connect (1) input control parameter nodes to action nodes, (2) call action nodes to output control parameter nodes, (3) assignment and call action nodes to action nodes, and (4) decision action nodes to action nodes. The transitions between decision action nodes and action nodes (the fourth item) are guarded by Boolean expressions in G.

- TO is a set of object transitions, defined as follows: $TO \subseteq (ON^l \times (AN^d \cup AN^a)) \cup (AN^a \times ON^l) \cup ((ON^{p,d} \cap ON^{p,i}) \times (AN^d \cup AN^a)) \cup (AN^a \times (ON^{p,d} \cap ON^{p,o}))$. That is, object transitions connect (1) local object nodes to decision and assignment action nodes, (2) assignment action nodes to local object nodes, (3) input data parameter nodes to decision and assignment action nodes, and (4) assignment action nodes to output data parameter nodes.

In our formalization of activity diagrams, we have excluded pseudo-nodes, namely the initial, final, fork, join, and merge nodes. Since the activity nodes in our formalization can have multiple incoming and outgoing transitions, pseudo-nodes do not lead to additional semantics. The semantics of our activity diagrams is identical to that described in [19]. Action nodes are the basic building blocks receiving inputs and producing outputs, called *tokens*. Tokens correspond to anything that flows through transitions. Tokens can be either data or control. Data tokens transit through object transitions (TO) and carry (partial) result values. Control tokens, however, transit through control transitions (TC) and carry null values.

Control tokens are meant to be used as triggers (events). Data tokens are associated with data parameter nodes and local object nodes, while control tokens are associated with control parameter nodes. An action node can start its execution only when *all* its input tokens via its incoming control or object transitions are provided. Upon its completion, an action node produces appropriate data and control tokens on its outgoing control and object transitions.

Assignment action nodes receive both control and data tokens as input and generate both data and control tokens as output (e.g., see Figure 3). Values are passed from one assignment action node to another via local object or parameter nodes. In other words, an assignment action node sends its output to a local object or a parameter node that is connected to the input of another action node. Call action nodes receive control tokens as input and produce control tokens as output (e.g., see Figure 6). Function calls are modeled by a call action node generating a token on an output control parameter node of the caller that is linked to some input control parameter node of the callee. Decision action nodes receive both control and data tokens as input, but generate only control tokens as output (e.g., see Figure 3).

Having described our formalization of activity diagrams, we now explain, using this formalization, how we account for block behaviors in the computation of estimated impact sets. The drawback of the algorithm presented earlier in Figure 7 is that it naively identifies all the output ports of any impacted block as impacted without regard to the intra-block dependencies between the impacted input ports and the output ports of that block. To address this drawback, we improve our algorithm as shown in Figure 8.

The modified algorithm of Figure 8 forward slices the activity diagrams related to the impacted blocks, starting from their impacted input ports. Recall from Figure 5 that each activity parameter node corresponds to some input port of a block related to that activity diagram. Our slicing criterion (i.e., the starting point) is described in terms of impacted input parameter nodes. In Figure 8, we use *correspond*(n) to denote the block ports related to an activity parameter node n, and use *correspond*$^{-1}(iP)$ to denote the set of activity parameter nodes related to the ports in iP.

Our forward static slicing algorithm, shown in Figure 9, is similar to existing forward program slicing approaches where slices are computed over *program dependency graphs* [7]. In these graphs, nodes correspond to program statements and are connected by edges representing control and data dependencies. The object transitions TO in our activity diagrams correspond to program def-use chains, specifying data dependencies [20]. The control transitions TC capture control dependencies from decision nodes to sequences of action nodes in the if-then-else branches as well as control dependencies between call action nodes and input/output control parameter nodes. The algorithm in Figure 9 computes, for

Algorithm. FORWARDSLICE.

Input: An activity diagram $ad = \langle AN, V, ON, G, TC, TO\rangle$.
A set *in* of *impacted* input parameter nodes of *ad* (slicing criterion).

Output: A set *iOut* of impacted output parameter nodes of *ad*.
A set *iAD* of impacted object and action nodes of *ad*.

1. $iOut = \emptyset$; $iAD = \emptyset$;
2. $R = TC \cup TO$
2. **for** $n \in in \cap ON^{p,i}$ **do**
3. $X = \{n\}$
4. **do**
5. $X = X \cup \{n' \mid \exists q \in X \cdot R(q, n') \vee \exists g \in G \cdot R(q, g, n')\}$
6. **until** X *reaches a fixed point*
7. $iAD = iAD \cup X$
8. $iOut = iOut \cup (X \cap ON^{p,o})$
9. **return** iN, $iOut$

Figure 9: Algorithm for activity diagram slicing (used by the algorithm of Figure 8).

any input parameter node n in the slicing criterion set (in), all the action and object nodes reachable from n via sequences of object and control transitions ($TC \cup TO$). The algorithm then returns all the reachable nodes (iAD) and all the reachable output parameter nodes ($iOut$).

For example, suppose that the slicing algorithm is called with ad set to the activity diagram of Figure 3, and in set to the Over-Temperature input parameter node. The algorithm computes for this activity diagram a forward slice such that: (1) the set iAD contains the decision node Temperature check, the assignment nodes Motor drive mode = OFF and Motor drive mode= ON, and the Motor drive mode output parameter node. And, (2) the set $iOut$ contains the Motor drive mode output parameter node which corresponds to an output port of block B_3 with the same label. Hence, the Motor drive mode output port of B_3 is the only output port that is likely to be impacted if a change is made to the Over-Temperature input port of B_3. Therefore, using the modified algorithm of Figure 8, we prune B_5, its ports, and its related behaviors from the *EIS*s computed for change requests Ch-R11 and Ch-R12.

An important remark about the computation of *EIS*s in our approach is that this process is meant to be *intertwined* with the implementation of a given change request. This intertwining provides a human feedback loop where the changes made to the models by the engineers at any given step is used for improving the accuracy of the *EIS* computed in the next steps. In particular, if the engineers modify the inter-block or intra-block dependencies in the SysML models during the implementation of a change request, the *EIS* needs to be recomputed. If no changes are made to the inter-block or intra-block paths, e.g., as is the case for Ch-R11 and Ch-R12 in our motivating example, the *EIS* will remain unaffected.

2.3 Ranking Potentially Impacted Elements

In the second step of our approach (marked 2 in Figure 4), we rank the elements of the *EIS* computed by the previous step. In addition to the *EIS*, this second step requires one or more natural-language statements from the engineers. These statements, which we call *change statements*, include the change description as well as any intuition that the engineers may have, based on their domain knowledge, about how a certain requirements change would propagate to the design. We denote the set of change statements by *chStat*.

Our ranking of the elements in the *EIS* is based on matching the natural-language labels of these elements against the keyphrases that appear in the statements of *chStat*. The keyphrases are extracted automatically using a keyphrase extractor. We use a tailored extractor that we developed in our previous work [6] for supporting requirements tasks. For example, applying the extractor over the statement "Temperature lookup tables and voltage converters need to be adjusted." given by the engineers for Ch-R11 (as discussed in Section 1) would identify the following keyphrases: "temperature lookup table" and "voltage converter".

With the keyphrases extracted, we use *similarity measures* to quantify how closely the text labels of the *EIS* elements match the keyphrases. Similarity measures can be *syntactic* or *semantic*. Syntactic measures are based on the string content of text segments (sometimes combined with frequencies). An example syntactic measure is *Levenshtein* [21], which computes a similarity score between two strings based on the minimum number of character edits required to transform one string into the other. Semantic measures are calculated based on relations between the meanings of words. An example semantic measure is *Path* [22], which computes a similarity score between two words based on the shortest path between them in an is-a hierarchy (e.g., an "automobile" is-a "vehicle" and so is a "train").

Similarity measures, both syntactic and semantic, are typically normalized to a value between 0 and 1, with 0 signifying no similarity and 1 signifying a perfect match. In line with common practice [22], we zero-out similarity scores below a certain threshold, in this paper 0.05, to minimize noise. In addition to individual similarity measures, we further consider pairwise combinations of syntactic and semantic measures due to these measures having a complimentary nature [16]. For the combination, we take the maximum of the two computed scores. Since there are several similarity measures to choose from, it is important to empirically investigate which measures are most suited to a specific task. Finding the best measures for our application context is addressed in our empirical evaluation (see Section 3).

Given an individual or a combined similarity measure, we compute for every $e \in EIS$ the similarity between the text label of e and the keyphrases obtained from *chStat*. The score we assign to e is the largest similarity score between e and any of the keyphrases. We then sort the elements of *EIS* in descending order of the scores assigned to the elements. The assumption here is that these scores are correlated with the likelihood of the elements being actually impacted by the change under analysis. In other words, we take the elements ranked higher in the sorted *EIS* to be more likely to be impacted. For example, for Ch-R12, we would obtain high scores for any block, port, activity node, or activity transition in the *EIS* that has a high degree of similarity to either "temperature lookup table" or "voltage converter".

3. EMPIRICAL EVALUATION

In this section, we investigate through an industrial case study the following Research Questions (RQs):

RQ1. (Usefulness of Slicing) *How much reduction in the size of EIS does our slicing technique bring about? Does slicing remove any actually-impacted elements (true positives) from the EIS computed by inter-block structural analysis?* With RQ1, we study the usefulness of our behavioral analysis by comparing the EISs obtained from structural analysis only (i.e., the algorithm of Figure 7) versus those obtained from both structural and behavioral analysis (i.e., the algorithm of Figure 8). In particular, we are interested in the magnitude of reductions in the EIS size that our behav-

ioral analysis provides without compromising the recall, i.e., without removing actually-impacted elements from the EIS.

RQ2. (Choice of Similarity Measures) *Which similarity measures are best suited to our approach?* There are several syntactic and semantic similarity measures for textual content. The choice of similarity measures used for calculating impact rankings directly affects the quality of our results. With RQ2, we identify the syntactic and semantic similarity measures that lead to the most accurate results.

RQ3. (Usability) *How should engineers use the ranked EIS lists produced by our approach?* For our approach to be useful, engineers need to determine how much of a ranked EIS list is worth inspecting. In other words, they need to determine a point in the list beyond which the remainder of the list is unlikely to contain impacted elements. With RQ3, we aim to develop systematic guidelines for inspecting the ranked EIS lists.

RQ4. (Effectiveness) *How effective is our automated approach when compared to a manual analysis performed by an engineer?* Assuming that the guidelines resulting from RQ3 are followed, RQ4 aims to determine whether our approach can reliably identify the set of actually impacted elements, and at the same time, save substantial inspection effort.

RQ5. (Scalability) *Does our approach have an acceptable execution time?* With RQ5, we study whether the execution time of our approach is practical.

Industrial Subject. Our case study is the cam phaser (CP) system introduced in Section 1. A SysML model for CP had been developed by the Delphi engineers in the Enterprise Architect tool [9]. This case study model consists of seven requirements diagrams containing 34 requirements, nine internal block diagrams with 48 blocks, 19 activity diagrams, and 56 traceability links, all of type `satisfy`. In total, the entire CP model contains 370 blocks, ports, and activity and object nodes. We chose CP as our case study model since it is an industrial system. The SysML model of CP contains a reasonable number of requirements and traceability links from requirements to blocks.

We were provided with 16 requirements change scenarios for CP. These scenarios are *real* and drawn from change requests originating from the customers of CP. In each case, a high-level natural-language statement was available which described the change as well as how the engineers expected the change to affect the design. One of these statements, referred to in our approach as a change statement, was illustrated in the motivating example of Section 1 (for Ch-R12). Five additional examples of change statements are provided in Table 1. As seen from the table, the statements are abstract and do not exactly pinpoint the impact of a change. Nevertheless, the keyphrases in the statements provide a mechanism for ranking the estimated impact sets computed by our approach. The actual impact set for each of the 16 change scenarios was further provided by the engineers involved in the case study.

Implementation. We have implemented our approach as a plugin for the Enterprise Architect modeling environment [9]. Our implementation enables users to automatically generate EISs using the algorithms of Section 2.2, and compute ranked EISs based on NLP similarity measures as discussed in Section 2.3. Our plugin is available at:

https://bitbucket.org/carora03/cia_addin

Table 1: Additional examples of change statements

id	Change Statements
1	Resolution of the battery voltage shall change from 0.1v to 0.01v. Battery voltage variables should be checked.
2	Input voltage divider of the battery voltage reading shall change from 0.2v to 0.3v. Measurement routines should be adjusted.
3	The motor control routine shall be executed in the 1ms task instead of the 2ms task. Slow regulator tasks should be revised.
4	Motor current shall increase from 45A to 50A. Shunt values and resolution of variables measuring the current should be revised.
5	The current mirror for the motor current measurement shall be replaced by a differential amplifier. Resolution settings should be revised.

Metrics. We use two well-known metrics, *precision* and *recall*, in our evaluation. Precision measures quality (i.e., low number of false positives) and is the ratio of actually-impacted elements found in an EIS to the size of the EIS. Recall measures coverage (i.e., low number of false negatives) and is the ratio of the actually-impacted elements found in an EIS to the number of all actually-impacted elements.

Results. Next, we discuss our RQs:

RQ1. (Usefulness of Behavioral Analysis) To answer this RQ, we compare the EISs generated by the two algorithms in Figures 7 and 8, using the CP SysML model and the 16 given change scenarios. Recall that the algorithm in Figure 7 relies on structural inter-block relations only, and the one in Figure 8 uses both structural inter-block and behavioral intra-block relations. We obtained 16 EISs via structural analysis and 16 other EISs via combined structural and behavioral analysis. We compared the sizes of the EISs, and their recall and precision values. The recall for all the 16 changes and for both the structural and the combined structural and behavioral approaches were 100%.

Figure 10 compares the EIS size and the EIS precision distributions for the 16 changes obtained by structural analysis and by the combined structural and behavioral analysis. The average EIS size and EIS precision based on structural analysis are, respectively, 80 and 8%, and based on the combined analysis are 38 and 16%, respectively. That is, after applying the forward slicing used in our behavioral analysis, on average, the EIS size is reduced by around 42 elements and the precision increases by 8%. We note that the total number of elements in the entire SysML model is 370. Hence, the EIS size generated by the structural analysis contains 21.6%, and the EIS size obtained by the combined analysis contains 9.7% of all the design elements. *In summary,* our results show that applying the combined structural and behavioral analysis significantly reduces the EIS size and increases precision without a negative effect on recall.

RQ2. (Choice of Similarity Measures for Ranking) To answer this RQ, we define a notion of accuracy for the ranked EISs (obtained from the ranking step in Section 2.3). We conceptualize accuracy using charts that show the percentage of actually-impacted elements identified (Y-axis) against the percentage of EIS elements traversed in the ranked list (X-axis). Figure 11 shows charts for an EIS, computed for one of the CP change scenarios, and ranked by two alternative applications of similarity measures.

A simple way to compare the alternatives would be the following: A similarity measure A (potentially, the combination of a syntactic and a semantic measure) is better than a measure B if the engineers are able to identify all the impacted elements by inspecting fewer elements when they use the list ranked by A than when they use the list ranked by B.

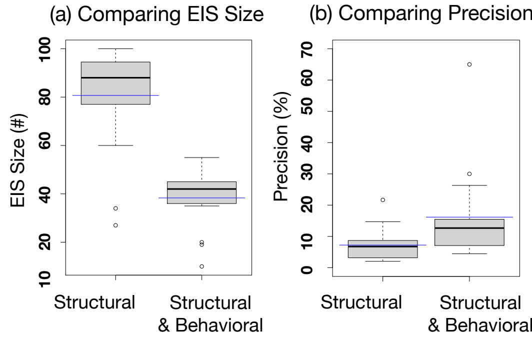

Figure 10: Impact of behavioral analysis: Comparing (a) the size of EISs and (b) the precision of EISs obtained by structural analysis alone and by combined behavioral and structural analysis.

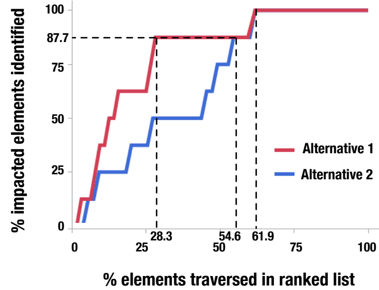

Figure 11: Two alternative applications of similarity measures for ranking the same EIS.

This intuition however, cannot distinguish the two alternatives in Figure 11, as both identify all the actually-impacted elements after traversing 61.9% of the elements.

Despite the above, Alternative 1 is better than Alternative 2 because it produces better results *earlier*. Specifically, if the engineers choose to stop, say after inspecting 30% of the list, with Alternative 1, they will find 87.7% of the actually-impacted elements. Identifying the same percentage of actually-impacted elements with Alternative 2 would require the inspection of $\approx 55\%$ of the list. To reward earlier identification of impacted elements, we use the Area Under the Curve (AUC) for evaluating accuracy. AUC can tell apart the two alternatives in Figure 11, as the metric is larger for Alternative 1.

We considered pairwise combinations of three syntactic measures, SoftTFIDF, Monge_Elkan and Levenshtein, from the SimPack library [23] and four semantic measures, LIN, PATH, RES and JCN, from the SEMILAR library [22]. We further considered alternatives where only a syntactic or only a semantic measure is applied. Specifically, we considered 19, i.e., $(4 + 1) \times (3 + 1) - 1$, alternatives. We ran these alternatives on our 16 changes, and obtained an AUC for each alternative.

To identify the best alternative for similarity measures, we need to determine which alternatives consistently result in the highest accuracy (i.e., AUC) when applied to the change scenarios. This analysis is often done using a regression tree [24], which is a hierarchical partitioning of a set of data points aimed at minimizing, with respect to a given metric, variations across partitions. In our context, each data point is a similarity measure alternative applied to an individual change scenario. We therefore have a total of $19 * 16 = 304$ data points. The metric of interest here is AUC.

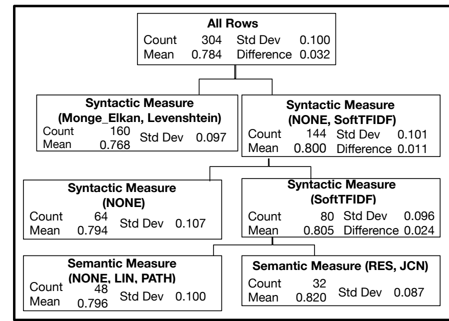

Figure 12: Regression tree for identifying the best similarity measure alternatives.

In Figure 12, we show the regression tree for our case study. In each node of the tree, we show the count (number of AUC values), the mean and standard deviation for the AUC values, the similarity measures generating the AUC values in that partition, and, for every non-leaf node, the difference between the mean AUC values of its right and left children. At each level, the factor (either syntactic measure or semantic measure) that best explains the variation in AUC values is selected. The partitioning at the first level of the tree signifies the most influential factor explaining the variation observed across the data points. In Figure 12, the most influential factor is the choice of syntactic measure.

The nodes on the right are of particular interest, as they signify the alternatives that result in higher AUC values on average than the alternatives of the left node. We iteratively partition the right-most nodes until the difference between the mean AUC values in the resulting branches is insignificant. We deem differences below 0.01 to be insignificant. At the first level, NONE and SoftTFIDF perform better than Monge_Elkan and Levenshtein. Depending on whether NONE appears in a syntactic or a semantic measure node, it indicates the stand-alone application of measures of the other type. For example, the NONE appearing alongside SoftTFIDF at the first level indicates stand-alone application of semantic measures. At the second level, the syntactic alternatives are further split suggesting that SoftTFIDF produces better results on average than NONE (i.e., stand-alone application of any semantic measure). The right-most node at the last level of the tree contains the most robust alternatives that yield the highest AUC values. The reason why SoftTFIDF performs best is because it filters noise: the measure assigns a zero (or very low) score to phrase pairs when their constituent tokens are not closely matching (lower than a certain threshold), or when the matching tokens are very common (e.g., stopwords).

In summary, and as suggested by the right-most leaf node in the tree of Figure 12, SoftTFIDF (syntactic measure) combined with either RES or JCN (semantic measures) would be the most suitable alternative. We use these two alternatives to answer RQ3 and RQ4.

RQ3. (Usability) To answer RQ3, we aim to find the best trade-off between the number of elements to inspect and the number of impacted elements found. We define a notion of *cutoff* indicating the percentage of a ranked list which is worthwhile inspecting, and hence, should be recommended to engineers. To define a cutoff, we use delta charts, as illustrated in Figure 13 for one of the change scenarios in CP. In the chart, at any position i on the X-axis, the Y-axis is the difference between the similarity scores at positions i

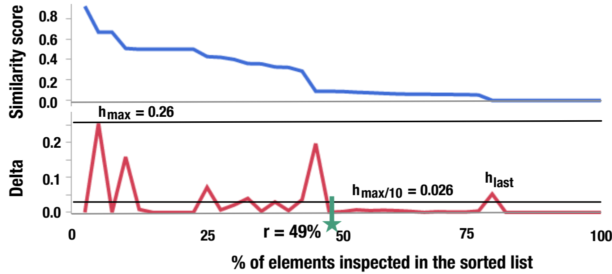

Figure 13: Ranked similarity scores and delta chart for an example change scenario from CP. The delta chart is used for computing the cutoff (r).

and $i-1$. For easier understanding, in Figure 13, we further show the ranked similarity scores on the top of the delta chart. These similarity scores were computed using SoftTFIDF (syntactic measure) and JCN (semantic measure). As described in Section 2.3, the label of each EIS element e is compared against all keyphrases in the change statement using both SoftTFIDF and JCN. The maximum value obtained from all these comparisons is assigned to e as its similarity score. The chart on the top of Figure 13 plots the EIS elements in descending order of the similarity scores.

For the cutoff, we pick the point on the X-axis after which there are no significant peaks in the delta chart. Intuitively, the cutoff is the point beyond which the similarity scores can no longer adequately tell apart the elements in terms of being impacted. What is a significant peak is relative. Based on our experiments, a peak is significant if it is larger than one-tenth of the highest peak in the delta chart, denoted h_{max} in Figure 13. The only exception is the peak caused by zeroing out similarity scores smaller than 0.05 (see Section 2.3). This peak, if it exists, is always the last one and hence denoted h_{last}. Since h_{last} is artificial in the sense that it is caused by zeroing out negligible similarity values, we ignore h_{last} when deciding about the cutoff.

More precisely, we define the cutoff r to be at the end of the right slope of the last significant peak (excluding h_{last}). In the example of Figure 13, $h_{max} = 0.26$. Hence, r is at the end of the last peak with a height $> h_{max}/10 = 0.026$. We recommend that engineers should inspect the EIS elements up to the cutoff and no further. In the example of Figure 13, the cutoff is at 49% of the ranked list. We note that the cutoff can be computed *automatically* and without user involvement. Therefore, the delta charts and their interpretation are transparent to the users of our approach.

In summary, for each change scenario, we automatically recommend, through the analysis of the corresponding delta chart as explained above, the fraction of the ranked EIS that the engineers should manually inspect for identifying actually-impacted elements.

RQ4. (Effectiveness) To answer RQ4, we report the results of applying the best similarity measure alternatives from RQ2 for ranking the EISs computed by the algorithm of Figure 8 (i.e., combined structural and behavioral analysis), and then considering only the ranked EIS fractions recommended by the guidelines of RQ3. Note that in this RQ, by EIS we mean the fraction obtained after applying the guidelines of RQ3. In Figure 14, we show for our 16 changes the size and precision distributions of the recommended EISs. These distributions are provided separately for the best similarity alternatives from RQ2, i.e., SoftTFIDF combined with

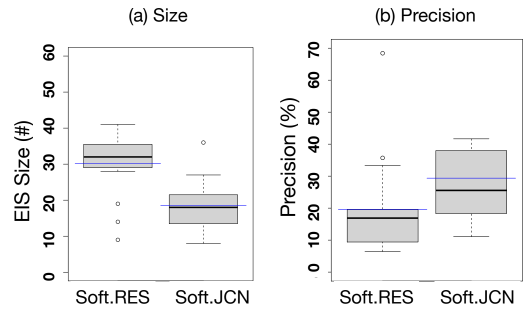

Figure 14: Size and precision of EISs that result from the application of the guidelines of RQ3 to the EISs computed by the algorithm of Figure 8.

RES (denoted Soft.RES) and SoftTFIDF combined with JCN (denoted Soft.JCN).

The average EIS size is 30.2 for Soft.RES and 18.5 for Soft.JCN. The average precision for Soft.RES and Soft.JCN are 19.5% and 29.4% respectively. As for recall, Soft.RES yields a recall of 100% for all 16 changes, while Soft.JCN misses one element for one change. That is, using Soft.JCN, we have a recall of 100% for 15 changes, and a recall of 85% for one change (i.e., an average recall of 99%). The results clearly show that Soft.JCN yields better overall accuracy.

In summary, after applying our best NLP-based similarity measure, Soft.JCN, the average precision of our analysis increases to 29.4% compared to 16% obtained by the combined behavioral and structural analysis (discussed in RQ1). The average recall reduces to 99% compared to 100% obtained by the combined analysis. Finally, using NLP, the average number of elements to be inspected by the engineers reduces to 18.5 (just 4.8% of the entire design model) compared to 38 (9.7% of the design model) before applying NLP.

RQ5. (Execution Time) The execution time for both steps of our approach, i.e., computing the EISs and ranking the EISs, was in the order of seconds for the 16 changes. Given the small execution times, we expect our approach to scale to larger systems. Execution times were measured on a laptop with a 2.3 GHz CPU and 8GB of memory.

Validity considerations and threats. Internal and external validity are the most relevant dimensions of validity for our case study. With regard to internal validity, an important consideration is that the change statements must represent the understanding of the engineers about a change *before* the engineers have determined the impact of that change; otherwise, the engineers may learn from the analysis they have performed and provide more precise change statements than when they have not examined the design yet. If this occurs, the accuracy results would not faithfully represent what one can achieve in a non-evaluation setting. In our case study, the change statements were pre-existing and written at the time that the change requests had been filed, i.e., before the impact of the changes had been examined. The engineers in our case study were therefore required only to inspect the design and provide the actual impact sets (gold standard). Consequently, learning is not a significant threat to internal validity. A potential threat to internal validity is that one of the engineers involved in our case study is a co-author. To minimize potential bias, the engineers involved neither used our tool nor saw the results generated by the tool until they had specified the actual impact sets. With regard to external validity, while our case study is in-

dustrial and we anticipate it to be representative of many embedded systems, particularly in safety-critical domains, additional case studies will be essential in the future.

4. RELATED WORK

There is a wide range of techniques for change impact analysis covering various development artifacts, including requirements, design and code [25]. Our work is concerned with analyzing how changes made to *requirements* will impact *system design*, in a context where the requirements are expressed using natural language, and the design using models. This situation is common and particularly relevant for embedded systems development. Below, we compare with the existing work that is most pertinent to the context in which we studied change impact analysis in this paper.

Any change impact analysis technique for requirements and design artifacts has to account for the dependencies between requirements and design elements to properly propagate changes. Aryani et al. [26] use logical relationships between domain concepts described in terms of weighted dependency graphs. van den Berg [27] augments traceability links with dependency type information between software artifacts. Goknil et al. [28] extend the approach of van den Berg [27] with formal semantics and apply it for impact analysis over requirements. When the requirements are expressed as models, more specialized dependency types may be defined. For example, Cleland-Huang et al. [29] use soft goal dependencies to analyze how changes in functional requirements propagate to non-functional requirements. Tang et. al. [30] capture dependencies using a special model, called an architectural rationale and linkage model, and use this model alongside probabilistic expert estimates for change impact analysis over system architectures.

Our work differs from the above in that we do not require dependency types, logical relationships between domain elements, architectural rationale, or probabilistic data for inferring or estimating impact likelihoods. Instead, we utilize the textual content of the design models and simple natural-language statements from engineers for impact likelihood prediction. The high expressiveness and implicit semantics of textual data make it difficult to come up with a crisp classification of dependencies or to obtain precise logical relations between the requirements and design. This is why we use (quantitative) similarity measures for predicting impact likelihoods.

A number of approaches rely on pre-defined rules for change propagation. Briand et al. [31] propose a taxonomy of model changes based on the UML metamodel, and use this taxonomy in order to specify rules for identifying which parts of a UML model need to be updated after each change so that the model will remain consistent with the UML metamodel. Müller and Rumpe [32] propose a domain-specific language for the specification of impact rules. These rules capture what kind of changes to models lead to what kind of impact. The main goal of these rule-based approaches is to maintain the consistency of models after changes; these approaches are not targeted at analyzing the impact of requirements changes. Our work has a different focus, as it aims at analyzing the impact of requirements changes on design. We do not use pre-defined impact rules in our approach.

Furthermore, there are approaches that rely on historical information obtained from software repositories for change propagation. For example, Wong and Cai [33] combine logical relationships between UML class diagrams and historical information obtained from software repositories to predict the scope of the impact of a given change. This approach relies on the existence of complete and consistent version histories. Such versioning is typically used for keeping track of changes in implementation-level artifacts, e.g., code. Our approach does not rely on historical data and can be used effectively in situations where no historical data is available, notably in early stages of development, or where detailed versioning of models is not practiced.

Control flow analysis techniques (e.g., software method call dependencies) have been used before to automatically identify traceability links [34], and further to facilitate code-level change impact analysis [35]. Kuang et al. [36] demonstrate that combining control flow and data dependencies improves automated retrieval of traceability links from requirements to code. Our results in this paper lead to a similar conclusion in the context of model-based development, that is, leveraging both inter-block data flow dependencies and intra-block control and data flow dependencies improves the accuracy of change impact analysis.

In our previous work [6], we already used similarity measures as change impact predictors. Nevertheless, this earlier work was focused on inter-requirement change impact analysis, i.e., identifying the impact of requirements changes on other *requirements*, rather than on the design. This earlier work focused exclusively on natural-language content. In our current work, we generalize our previous work to a model-based development setting, where we exploit not only the natural-language content of the development artifacts but also the structure and semantics of the design models.

5. CONCLUSION

We presented an approach to automatically identify the impact of requirements changes on system design. Our approach has two main steps: First, for a given change, we obtain a set of estimated impacted model elements by computing reachability over inter-block data flow and intra-block control and data flow dependencies. Next, we rank the resulting set of elements according to a quantitative measure obtained using NLP techniques. The measure reflects the similarity between the textual content of the elements in the estimated impact set and the keyphrases in the engineers' statements about the change.

Our evaluation on an industrial system shows that the accuracy of our approach consistently improves when we consider both inter-block and intra-block dependencies rather than only the inter-block ones, and the textual content of the diagrams in addition to only the elements' dependencies. Although not included in this paper, we note that eliminating the analysis in the first step of our approach and only applying the NLP technique in the second step reduces the accuracy considerably. This is because many elements in parts of the design unreachable for a given change have some degree of textual similarity with the change statements. In the future, we intend to make the change statements more structured, e.g., by introducing a controlled natural language. This can make change impact analysis more targeted and deal with more complex situations.

6. ACKNOWLEDGEMENT

We gratefully acknowledge funding from Delphi Automotive and from Fonds National de la Recherche, Luxembourg under grant FNR/P10/03 - Verification and Validation Lab.

7. REFERENCES

[1] Arnold, R., Bohner, S.: Software Change Impact Analysis. Wiley-IEEE Computer Society Press (1996)

[2] Pfleeger, S.L., Atlee, J.M.: Software engineering - theory and practice (4. ed.). Pearson Education (2009)

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

[4] : Road vehicles – functional safety. ISO draft standard (2009)

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

[6] Arora, C., Sabetzadeh, M., Goknil, A., Briand, L.C., Zimmer, F.: Change impact analysis for natural language requirements: An NLP approach. In: 23rd IEEE International Requirements Engineering Conference, RE 2015, Ottawa, ON, Canada, August 24-28, 2015. (2015) 6–15

[7] Tip, F.: A survey of program slicing techniques. J. Prog. Lang. **3**(3) (1995)

[8] Jurafsky, D., Martin, J.H.: Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. 1st edn. Prentice Hall PTR (2000)

[9] : Enterprose Architect (EA). http://www.sparxsystems.com.au/

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

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

[12] Cleland-Huang, J., Gotel, O., Hayes, J.H., Mäder, P., Zisman, A.: Software traceability: trends and future directions. In: Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014. (2014) 55–69

[13] Rempel, P., Mäder, P., Kuschke, T., Cleland-Huang, J.: Mind the gap: assessing the conformance of software traceability to relevant guidelines. In: 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014. (2014) 943–954

[14] Mäder, P., Jones, P., Zhang, Y., Cleland-Huang, J.: Strategic traceability for safety-critical projects. IEEE Software **30**(3) (2013) 58–66

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

[16] Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S.M., Zave, P.: Matching and merging of variant feature specifications. IEEE Trans. Software Eng. **38**(6) (2012) 1355–1375

[17] Briand, L.C., Falessi, D., Nejati, S., Sabetzadeh, M., Yue, T.: Traceability and SysML design slices to support safety inspections: A controlled experiment. ACM Trans. Softw. Eng. Methodol. **23**(1) (2014) 9:1–9:43

[18] Sabetzadeh, M., Nejati, S., Briand, L.C., Mills, A.E.: Using SysML for modeling of safety-critical software-hardware interfaces: Guidelines and industry experience. In: 13th IEEE International Symposium on High-Assurance Systems Engineering, HASE 2011, Boca Raton, FL, USA, November 10-12, 2011. (2011) 193–201

[19] Bock, C.: SysML and UML 2 support for activity modeling. Systems Engineering **9**(2) (2006) 160–186

[20] Alomari, H.W., Collard, M.L., Maletic, J.I.: A very efficient and scalable forward static slicing approach. In: 19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012. (2012) 425–434

[21] Manning, C.D., Raghavan, P., Schütze, H.: Introduction to information retrieval. Cambridge University Press (2008)

[22] Rus, V., Lintean, M.C., Banjade, R., Niraula, N.B., Stefanescu, D.: SEMILAR: the semantic similarity toolkit. In: 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, Proceedings of the Conference System Demonstrations, 4-9 August 2013, Sofia, Bulgaria. (2013) 163–168

[23] : the SimPack library. http://www.ifi.uzh.ch/ddis/simpack.html

[24] Breiman, L., Friedman, J., Olshen, R., Stone, C.: Classification and Regression Trees. Wadsworth and Brooks, Monterey, CA (1984)

[25] Lehnert, S.: A review of software change impact analysis. Technical Report ilm1-2011200618, Technische Universitat Ilmenau (2011)

[26] Aryani, A., Peake, I., Hamilton, M.: Domain-based change propagation analysis: An enterprise system case study. In: 26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania. (2010) 1–9

[27] van den Berg, K.: Change impact analysis of crosscutting in software architectural design. In: Workshop on Architecture-Centric Evolution (ACE 2006), Groningen (July 2006) 1–15

[28] Goknil, A., Kurtev, I., van den Berg, K., Spijkerman, W.: Change impact analysis for requirements: A metamodeling approach. Information & Software Technology **56**(8) (2014) 950–972

[29] Cleland-Huang, J., Settimi, R., Benkhadra, O., Berezhanskaya, E., Christina, S.: Goal-centric traceability for managing non-functional requirements. In: Proceedings of the 27th International Conference on Software Engineering. ICSE '05 (2005) 362–371

[30] Tang, A., Nicholson, A., Jin, Y., Han, J.: Using bayesian belief networks for change impact analysis in architecture design. J. Syst. Softw. **80**(1) (jan 2007) 127–148

[31] Briand, L., Labiche, Y., O'Sullivan, L.: Impact analysis and change management of UML models. In: Proceedings of the 19th IEEE International Conference on Software Maintenance (ICSM '03). (2003) 256–265

[32] Müller, K., Rumpe, B.: A model-based approach to impact analysis using model differencing. Electronic Communications of the EASST **65** (2014)

[33] Wong, S., Cai, Y.: Predicting change impact from logical models. In: 25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada. (2009) 467–470

[34] Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. Journal of Software: Evolution and Process **25**(1) (2013) 53–95

[35] Li, B., Sun, X., Leung, H., Zhang, S.: A survey of code-based change impact analysis techniques. Softw. Test., Verif. Reliab. **23**(8) (2013) 613–646

[36] Kuang, H., Mäder, P., Hu, H., Ghabi, A., Huang, L., Lv, J., Egyed, A.: Do data dependencies in source code complement call dependencies for understanding requirements traceability? In: Proceedings of 28th IEEE International Conference on Software Maintenance (ICSM'12). (2012) 181–190