Heterogeneous Megamodel Slicing for Model Evolution

Rick Salay
Department of Computer Science
University of Toronto
Toronto, Canada
rsalay@cs.toronto.edu

Sahar Kokaly
McMaster Centre for Software Certification
McMaster University
Hamilton, Canada
kokalys@mcmaster.ca

Marsha Chechik
Department of Computer Science
University of Toronto
Toronto, Canada
chechik@cs.toronto.edu

Tom Maibaum
McMaster Centre for Software Certification
McMaster University
Hamilton, Canada
maibaum@mcmaster.ca

ABSTRACT

Slicing is a widely used technique for supporting comprehension and assessing change impact during software evolution activities. While there has been substantial research into the slicing of particular model types, model-based software development typically involves heterogeneous collections of related models and there is little work addressing slicing in this context. In this paper, we propose a generic slicing approach for "megamodels" – a well-known model management technique for representing and manipulating collections of models and relationships between them. Our approach exploits existing model slicers for particular model types as well as the traceability relationships between models to address the broader heterogeneous model slicing problem. We illustrate our approach on an example of evolution in model-based automotive software development.

Keywords

Evolution, model slicing, model management, megamodels.

1. INTRODUCTION

Slicing is a widely used technique for supporting software evolution activities [19]. Specifically, *static* slicing [26] can identify the subset of software that is semantically dependent on a specific portion that has or is planned to be changed and hence is useful for assessing change impact due to evolution. In the MDE context, model slicing has been studied for particular model types, e.g., State Machines [16, 18], Class Diagrams [13, 18], etc. However, large-scale software systems are often described using heterogeneous collections of interrelated models, and change impact analysis requires a broader slicing approach that can address this.

While some work has addressed slicing for heterogeneous model collections, these have been limited to a specific set of model types (e.g., [20]) or remain at a theoretical level (e.g., [7]). In this paper, we propose a general and pragmatic static slicing algorithm for heterogeneous model collections. Specifically, (1) it operates on "megamodels" – a general modeling technique to represent collections of interrelated models; (2) it can work with arbitrary model types (e.g., conceptual, behavioural, goal models, test models, etc.) by utilizing their corresponding type-specific model slicers; and (3) it uses the widely adopted notion of traceability relations to assess change impact between models. We then analyze the proposed algorithm for termination, correctness, running time and minimality.

The remainder of this paper is structured as follows. In Sec. 2, we give a motivating example from the automotive software domain. In Sec. 3, we recall the background needed for the slicing approach and in Sec. 4, we describe the proposed slicing algorithm and its analysis. Then, in Sec. 5, we give a detailed illustration of the algorithm on the automotive example. In Sec. 6, we discuss related work and finally, in Sec. 7, we give our conclusions and report on future work.

2. MOTIVATING EXAMPLE

Consider an automotive subsystem that controls the behaviour of a power sliding door in a car. The system has an `Actuator` that is triggered on demand by a `Driver Switch`. This example is presented in Part 10 of the ISO 26262 standard [12]. Refer to Fig. 1 which shows the system models comprised of a Class Diagram (to model its structure), a Sequence Diagram (to model its behaviour) and a relationship between them. This can be visualized at a high-level as the megamodel (to be defined in Sec. 3) in Fig. 2.

The `Driver Switch` input is read by a dedicated Electronic Control Unit (ECU), referred to as `AC ECU`, which powers the `Actuator` through a dedicated power line. The vehicle equipped with the item is also fitted with an ECU which is able to provide the vehicle speed (VS). This ECU is referred to as `VS ECU`. The system includes a safety element, namely, a `Redundant Switch`. Including this element ensures a higher level of integrity for the overall system.

The `VS ECU` control unit provides the `AC ECU` with the vehicle speed. The `AC ECU` monitors the driver's requests, tests if the vehicle speed is less than or equal to 15 km/h, and if so, commands the `Actuator`. Thus, the sliding door can only be opened or closed if the vehicle speed is no more than 15 km/h. The `Redundant Switch` is located on the power line between the `AC ECU` and the `Actuator` as a secondary safety control. It switches on if the speed is less than or equal to 15 km/h, and off whenever the speed is greater than 15 km/h. It does this regardless of the state of the power line

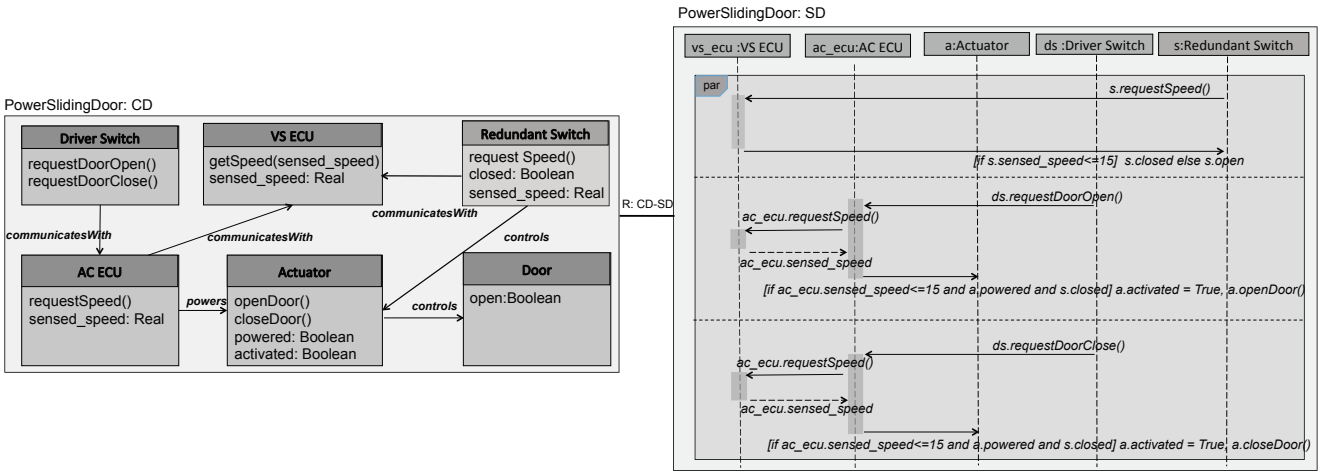

Figure 1: Power sliding door system models.

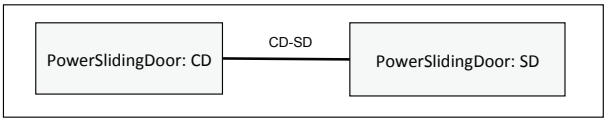

Figure 2: Power sliding door system megamodel.

(its power supply is independent). The `Actuator` operates only when it is powered.

Now, consider that the power sliding door system changes and the redundant switch is removed. This could be due to the need to minimize cost and produce a cheaper vehicle. In the new system, only the `AC ECU` checks the vehicle speed before commanding `Actuator`. In this case, it would be desirable to provide a sliced megamodel of the system that reflects the parts of the original megamodel affected by this change in order to help with system evolution activities. For example, we shown that with safety-critical software, such as for automotive systems, a system megamodel slice is an essential part of re-assessing the safety assurance of the system [15].

After presenting our slicing approach, we demonstrate it on the power sliding door example in Sec. 5.

3. BACKGROUND AND PRELIMINARIES

3.1 Megamodeling and Model Management

A complexity problem in MDE arises due to the proliferation of software models, and the area of Model Management [2] has emerged to address this challenge. Model management focuses on a high-level view in which entire models and their *relationships* (i.e., mappings between models) can be manipulated using *operators* (i.e., specialized model transformations) to achieve useful outcomes. In this paper, we focus on one of the model management operators – model *slice* [20]. Other model management operators that have been studied include *match* [2], *diff* [2], *lift* [23], and *merge* [6]. Each of these model management operators can be viewed as an *abstract* transformation that defines a class of concrete transformations, i.e., the implementations that refine the operator for particular model types. For example, a slice operator for class diagrams is implemented differently than a slice operator for state machines.

Megamodels. To help visualize and work with collections of models and their relationships, model management uses a special type of model called a *megamodel* [3]. In this paper, we define this and related concepts as follows.

Definition 1 (Megamodel). *A* megamodel *is a model with elements representing models and links between elements representing relationships between the models.*

Fig 3 shows the simplified metamodel used for megamodels in this paper. A `Megamodel` consists of a graph of named and typed `Model` elements with `Relationship` links connecting them. These refer to models and model relationships (defined below), respectively, and the types indicate their metamodels. The well-formedness constraint requires that the models on either end of every relationship are distinct. We have made the simplifying assumptions that (1) all relationships are binary; and (2) megamodels cannot be nested or reference other megamodels. In Sec. 4.3, we discuss how are these assumptions can be relaxed.

Fig. 2 shows a megamodel for our power sliding door example. Here, `PowerSlidingDoor : CD` and `PowerSliding−Door : SM` refer to the Class Diagram and Sequence Diagram, respectively, while the line connecting them refers to the relationship of type `CD − SD` for connecting these two types of models. Note that relationship names are optional.

Definition 2 (Model). *A* model *is set of typed elements and links conforming to a metamodel. We use the term* atom *to denote either an element or a link.*

Definition 3 (Model relationship). *A* model relationship *connecting models M and M' consists of a set of typed links conforming to a metamodel. Each link connects atoms of M to atoms of M'.*

Definition 4 (Traceability relationship). *A* traceability relationship *is a model relationship in which the links express a* dependency *relationship between the atoms it connects. The dependency relationship can be unidirectional or bidirectional depending on the type of traceability link.*

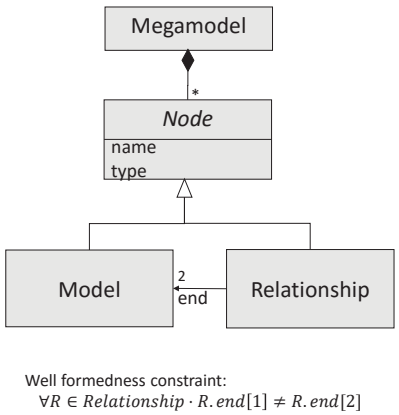

Figure 3: (Simplified) metamodel for megamodels.

Note that the definition of traceability relationship used in this paper is broader than that typically used by requirements engineering [11] and narrower than what is sometimes used for general modeling [1]. We focus solely on traceability relationships and use them to determine cross-model dependencies. In fact, we assume that the only relationships in the megamodels are the traceability relationships. In Sec. 4.3, we discuss how this assumption can be relaxed.

Definition 5 (Model fragment). *A* model fragment *S of a model M, denoted $S[M]$, is any subset of atoms of M.*

Definition 6 (Megamodel fragment). *A* megamodel fragment *S of a megamodel X, denoted $S[X]$, is any set of model fragments of the models in X. We say that $S[X]$ is contained in $S'[X]$, denoted $S[X] \sqsubseteq S'[X]$, iff the following condition holds:*

$$\forall M \in X \cdot \cup\{S[M] | S[M] \in S[X]\} \subseteq \cup\{S'[M] | S'[M] \in S'[X]\}$$

Thus, $S[X] \sqsubseteq S'[X]$ when, for each model $M \in X$, the combined model fragments of M in $S[X]$ is contained in the combined model fragments of M in $S'[X]$. Note that a megamodel fragment is defined as containing only model fragments and no relationships between them.

3.2 Model slicing

Program slicing, and, correspondingly, model slicing approaches, fall into four categories: *static*, *dynamic*, *conditional* and *amorphous* [7]. In each case, we are given a model and a slicing criterion indicating some "aspect of interest" in the model, and the slicing process produces a slice of the model that addresses the criterion. Static slicing uses a model fragment as the criterion. A *forward slice* expands the criterion to all dependent atoms while a *backward slice* expands to all depending atoms. While static slicing uses a subset of the syntax as a criterion, dynamic slicing uses a constraint from the semantic domain. For example, dynamic slicing can be used to identify the classes used in a particular run of a program. Conditional slicing combines both static and dynamic approaches by allowing a hybrid criterion. Finally, while the first three types of slicing produce a slice that is a fragment of the model, amorphous slicing allows the slice to be a different model. For example, the approach used in [20] adds stuttering transitions to state machine slices in order to preserve behaviours.

In this paper, we focus on static forward slicing since it is readily applicable to assessing the impact of changes due to model evolution. We define this as follows.

Definition 7 (Static forward model slice). *Given a model M and model fragment $S[M]$, the* static forward slice *of M with respect to the slicing criterion $S[M]$ is the model fragment $S'[M]$ satisfying the following conditions:*

1. *(Correctness) $S'[M]$ contains all atoms of M that are directly or indirectly dependent on atoms of $S[M]$.*
2. *(Minimality) Every atom of $S'[M]$ is either directly or indirectly dependent on atoms of $S[M]$.*

Note that since $S[M]$ is dependent on itself, these conditions imply that $S[M] \subseteq S'[M]$.

4. MEGAMODEL SLICING

In this section, we propose a slicing approach for heterogeneous megamodels. Intuitively, such a slicer should allow the criterion to be expressed as a megamodel fragment and the forward slice should expand this to the megamodel fragment containing all dependent elements. We generalize Def. 7 to capture this intuition.

Definition 8 (Static forward megamodel slice). *Given a megamodel X and megamodel fragment $S[X]$, the* static forward slice *of X with respect to slicing criterion $S[X]$ is the megamodel fragment $S'[X]$ satisfying the following conditions for all $M \in X$:*

1. *(Correctness) There exists a model fragment $S'[M] \in S'[X]$ that contains all atoms of M that are directly or indirectly dependent on the atoms of any model fragment in $S[X]$.*
2. *(Minimality) If there exists a model fragment $S'[M] \in S'[X]$, then every atom of $S'[M]$ is either directly or indirectly dependent on the atoms of some model fragment in $S[X]$.*

There are two levels of expansion in this slicing process: (1) expansion within individual models to dependent elements and, (2) expansion between models across relationships to dependent elements in neighbouring models. This two-level process is repeated until it produces no further expansion. For (1), we leverage existing type-specific slicers that take the semantics of the individual model types into account. For (2), we use the links in traceability relationships to connect dependent elements. Here, no special relationship-type specific slicers are needed since all relationship types are assumed to be sets of links and every link is assumed to represent a dependency.

Note that this definition of slicing is a *deep* slicing since the process includes the content of the models and relationships referenced by the elements of the megamodel. In contrast, a *shallow* megamodel slicing would be one that only considered the elements of the megamodel and not what they reference. Here, a subset of a megamodel (the criterion) is expanded to the subset that is connected directly or indirectly via relationship links (the slice), i.e., the shallow slice is the largest subset contained in the transitive closure of the initial subset taken along relationship links. There may be some use cases in which shallow megamodel slicing is useful but in this paper we focus on the deep version.

4.1 Slicing algorithm

Fig. 4 gives the algorithm for forward slice. The input is megamodel X with megamodel fragment $S_c[X]$ given as the slicing criterion. The output is megamodel fragment $S[X]$ representing the forward slice. The algorithm makes the following assumptions:

ASSUMPTION 1. *For each model type T represented in X, we have a slicer* $\texttt{Slice}_T$ *for models of type T that satisfies Def. 7.*

ASSUMPTION 2. *The set of traceability relationships in X express all and only the direct dependencies between atoms of models in X.*

In addition, we require several simple supporting operations.

DEFINITION 9 (`Union`). *Given a pair of megamodel fragments $S_1[X], S_2[X]$, the* megamodel fragment union*, denoted* $\texttt{Union}(S_1[X], S_2[X])$*, is defined with the following condition.*

$$\forall S[M] \in \texttt{Union}(S_1[X], S_2[X]) \cdot$$
$$S[M] = \cup\{S'[M] | S'[M] \in S_1[X] \cup S_2[X]\}$$

Thus, the $\texttt{Union}(S_1[X], S_2[X])$ can be constructed by first taking the set union $S_1[X] \cup S_2[X]$ and then unioning all model fragments of the same model within this.

DEFINITION 10 (`Trace`). *Given a traceability relationship R with ends M and M', and model fragment $S[M]$, the* trace *of $S[M]$ along R, denoted* $\texttt{Trace}(R, S[M])$ *is the model fragment $S'[M']$ consisting of the subset of atoms in M' dependent on the atoms in M according to R.*

We compute $\texttt{Trace}(R, S[M])$ by following the links of R from the atoms of M to the atoms of M'.

DEFINITION 11 (`OppEnd`). *Given a traceability relationship R with ends M and M', we define* $\texttt{OppEnd}(R, M) = M'$ *and* $\texttt{OppEnd}(R, M') = M$.

In line 1 of the algorithm, the current slice is initialized to the criterion. The two levels of expansion are in lines 4-9 and lines 10-17, respectively, inside the main loop of lines 2-19. For level 1, the temporary result $S_1[X]$ is initialized in line 3 to the empty set and then lines 5-9 iterate through the model fragments in the current slice. In line 7, the model type-specific slice is computed using the model fragment as the criterion and the result is accumulated in $S_1[X]$ (line 8).

The level 2 expansion temporary result $S_2[X]$ initialized on line 10. The outer iteration (lines 11-17) is over the model fragments from the level 1 expansion, and the inner iteration (lines 12-16) is over each relationship connected to the model fragment. Note that the set of relationships connected to a model fragment $S_1[M]$ is the set of relationships connected to M via the `end` property (see Fig. 3). For each such relationship R, we first determine the model M' on the other end of the relationship using supporting function `OppEnd` in line 13. Then in line 14, the model fragment $S_2[M']$ is produced by tracing the links in R from $S_1[M]$ to M'. Finally, in line 15, this result is accumulated in $S_2[X]$.

Algorithm: Forward Megamodel Slice
Input: megamodel X, criterion megamodel fragment $S_c[X]$
Output: slice megamodel fragment $S[X]$

```
1:  S[X] := S_c[X]
2:  do {
3:      S'[X] := S[X]
4:      S_1[X] := ∅
5:      for (S[M] ∈ S[X]) {
6:          T := M.type
7:          S_1[M] := Slice_T(M, S[M])
8:          S_1[X] := Union(S_1[X], {S_1[M]})
9:      }
10:     S_2[X] := ∅
11:     for (S_1[M] ∈ S_1[X]) {
12:         for (R ∈ M.end) {
13:             M' := OppEnd(R, M)
14:             S_2[M'] := Trace(R, S_1[M])
15:             S_2[X] := Union(S_2[X], {S_2[M']})
16:         }
17:     }
18:     S[X] := Union(S_1[X], S_2[X])
19: } until (S[X] ⊑ S'[X])
20: return S[X]
```

Figure 4: Algorithm for forward megamodel slice.

After the two levels of expansion, the combined result is computed in line 18 and checked to see if any actual expansion has occurred (line 19). If no expansion has occurred, a fixed point has been reached and the main loop exits with the current slice returned as the final result in line 20; otherwise, the main loop repeats.

4.2 Analysis

We consider the issues of termination, complexity and correctness for forward slice algorithm in Fig. 4.

Termination. We show that the slicing algorithm is guaranteed to terminate. After the level 1 expansion loop completes (lines 5-9), it is clear that $S'[X] \sqsubseteq S_1[X]$ since $S_1[X]$ is constructed by expanding each model fragment in the current slice $S[X]$ using type-specific slicers (see Assumption 1) and doing `Union` (see Def. 9). Furthermore, $S'[X] = S[X]$ (line 3). Then, in line 18, when the new slice is computed, $S_1[X] \sqsubseteq S[X]$ since `Union` cannot produce a result smaller than its arguments. Therefore, $S'[X] \sqsubseteq S[X]$. Thus, on line 19, either no expansion has occurred ($S[X] \sqsubseteq S'[X]$) and the algorithm terminates or some expansion has occurred and the loop iterates again. Thus, in each iteration, the current slice can only get larger and since this process is bounded by X, the algorithm must terminate.

Time Complexity. The level 1 loop (lines 5-9) can iterate N_M times and the level 2 loop (lines 11-17) can iterate N_M^2 times where N_M is the number of models in X. The dominating operation in the level 1 loop is the type-specific slicer. Since the time complexity varies according to the slicer used, we represent it using a type-independent upper bound $SL(n)$ as a function of the number of elements n in the input model. Tracing along a relationship and union (lines 14-15) is $O(N_a)$ in the worst case, where N_a is the total number of atoms across all models of X. Thus, in the worst case, one iteration of the main loop is $O(N_M \times SL(N_a) + N_M^2 \times N_a)$.

Finally, in the worst case, the size of the current slice can increase by one in each iteration of the main loop, for N_a iterations. Thus, the time complexity is given by:

$$O(N_a \times N_M \times SL(N_a) + N_M^2 \times N_a^2)$$

Correctness. We argue that the slicing algorithm satisfies the correctness condition in Def. 8. Assume that the algorithm is at line 3 and there exists a non-empty set of atoms not in the current slice $S[X]$ that are dependent on atoms of model fragments in $S[X]$. Note that if some atom a is indirectly dependent on an atom a', then there must be a sequence of directly dependent atoms $a, a_1, ..., a_n, a'$ connecting them. Thus, there must also be a non-empty set of atoms not in $S[X]$ that are *directly* dependent on atoms of model fragments in $S[X]$. Let us choose one such atom a' in some model M' in X that is directly dependent on an atom a in some model fragment $S[M]$ in $S[X]$. We consider the two cases: $M' = M$ and $M' \neq M$.

Case 1). If $M' = M$, then by Assumption 1, the slicer used in line 7 satisfies the correctness condition in Def. 7 and thus, atom a' will be added to a model fragment in $S_1[X]$ in an iteration of the level 1 loop (lines 5-9).

Case 2). If $M' \neq M$, then by Assumption 2, there is a traceability relationship R in X with a link that connects a to a' and thus, line 14 will cause a' to be added to a model fragment in $S_2[X]$ in an iteration of the level 2 loop (lines 11-17).

In either case, the atom a' will enter the next iteration of the slice in line 18. Furthermore, since the addition of a' expands the slice, the main loop will iterate again and will capture the next set of directly dependent atoms, and so on. When the set of directly dependent atoms not in $S[X]$ is empty, no further level 1 or level 2 expansion is possible, and the algorithm terminates.

Minimality. We show that the slicing algorithm satisfies the minimality condition in Def. 8. To do this, we must show that the slice produced by the algorithm contains no atom that is not dependent on the criterion. Assume that there is an atom a' in the final slice that is not dependent on the criterion. In this case, a' must have been added to the slice on line 7 or line 14 in some iteration of the main loop. However, by Assumption 1 and Def. 7, $\texttt{Slice}_T$ can only produce minimal model slices in line 7 and so a' could not have been added there. Also, by Assumption 2, traceability relationships only contain links between true dependencies and in line 14, `Trace` is applied from the current slice to these dependent atoms. Thus, a' could not have been added at line 14. Therefore, we have a contradiction and so the megamodel slice must be minimal.

4.3 Discussion

Well-formedness and referential integrity. Def. 7 does not require that a slice be a well-formed model. However, in practice, ensuring that a slice is well-formed may be desirable because the slice can be used directly by tools such as editors, analyzers and transformations. Making a model fragment into a well-formed model requires it to be expanded by a minimum number of atoms in order to satisfy the well-formedness constraints. For example, if a CD fragment contains an association without one of its endpoints, adding the missing endpoint class will make it well-formed.

The problem with doing this expansion is that atoms can be added that are *not dependent on the criterion* since, if they were dependent, then they would already be in the slice. In particular, if $\texttt{Slice}_T$ used in line 7 of the slicing algorithm always included an expansion to well-formedness then in the subsequent steps of the algorithm the atoms added for well-formedness would be treated as though they were atoms added for dependency. This would result in a non-minimal megamodel slice. As a result, we view the expansion to well-formedness as an *optional post-processing step* that could be applied after the megamodel slice is computed.

A similar argument can be made about the issue of *referential integrity*. Assume that one atom references another, e.g., a lifeline in a sequence diagram references the class of the object that the lifeline represents. The referenced class is not dependent on the referencing lifeline; thus, if the forward slice includes the lifeline, it need not contain the class. However, it may be desirable to expand the slice to include the class to provide relevant contextual information for the lifeline. As with well-formedness, this referential integrity expansion can introduce atoms that are not dependent on the criterion and thus such an expansion should only be done as a post-processing step on the slice.

Generalizing the slicing algorithm. In Sec. 3, we made several simplifying assumptions in order to focus on the core aspects of the slicing algorithm. We now briefly discuss how to relax these assumptions.

- **N-ary Relationships.** We have assumed that all relationships in the megamodel are binary but it is straightforward to extend the algorithm to handle N-ary relationships. Specifically, the iteration through the relationships (lines 12-16) must be generalized to handle the case where a traceability link holds between atoms in models on multiple `end`s, and the supporting operations `OppEnd` and `Trace` must be adapted to address this.
- **Nested megamodels.** In the general case, a megamodel can contain other megamodels. Such a megamodel could be viewed as a tree with models as leaves and nested megamodels as intermediate nodes. A megamodel fragment is a tree with the same structure but with model fragments as leaves. Thus, the algorithm follows a similar approach as currently but in addition it must preserve the megamodel tree structure in the final slice.
- **Arbitrary relationships.** We have assumed that all relationships are traceability relationships since these are the only ones that matter to the slicing algorithm. In general, however, there may be other types of relationships in the megamodel, e.g., refinement, overlap, etc. The simplest way to allow these relationship types is to ignore all non-traceability relationships in the loop in lines 12-16.

5. POWER SLIDING DOOR EXAMPLE

In this section, we demonstrate our slicing approach on the power sliding door example presented in Sec. 2.

5.1 Megamodels of class and sequence diagrams

For the purpose of the example presented here, we instantiate our general framework such that its input is a system megamodel X given by a class diagram `CD`, a sequence diagram `SD`, and a relationship $\texttt{CD} - \texttt{SD}$ between them. Note that, although these are both UML diagrams, we are treat-

Table 1: Dependency relations for CD and SD slicers.

Rule	Component under assessment	Dependant parts potentially impacted
CD1	Class	Owned attributes and methods. Associations connected to class. Attributes/methods in other classes using types introduced in this class. Subclasses.
SD1	Term (portion of an expression)	Associated expression.
SD2	Expression (guard/action)	Associated message.
SD3	Message	Associated arrow (from source to target lifeline).
SD4	Arrow	Arrows directly after the arrow in the sequence. Message on the arrow.
SD5	Lifeline	Arrows connected to the lifeline. Messages on arrows connected to the lifeline.

ing them separately for the sake of this example. In general, not all models in a megamodel have to be UML diagrams.

Assume we are given some known change on the megamodel, which represents the slicing criterion $S_c[X]$ used as input to our algorithm. As stated in Sec. 4, we also assume that we are provided with correct class diagram and sequence diagram model slicers similar to those presented in [18] and [21], respectively.

For simplicity, we define our own CD and SD slicers for this example as follows:

• CD slicer works with the dependency rule shown as CD1 in Table 1: If a class is being considered for impact assessment, then all of its attributes, methods, associations linked to it and its subclasses are considered dependant on it and could potentially be impacted. They are therefore to be added in the slice.

• SD slicer works with the dependency rules shown as SD1 − SD5 in Table 1: If a term, i.e., any portion of an expression (e.g., a guard or an action) in a message, is being considered for impact assessment, then its associated expression could be impacted. Similarly, if an expression (e.g., a guard or an action) is being considered, its associated message should be included in the slice. Other rules for impact assessment of messages, arrows and lifelines are shown in the table.

Note that both slicers satisfy Def. 7, i.e., they are correct and minimal. We also assume that the set of traceability relationships in CD − SD expresses all and only the dependencies between the CD and SD in our system megamodel.

5.2 Slicing of Power Sliding Door megamodel

Recall the Power Sliding Door megamodel presented in Fig. 2 which we refer to as PSD. The models represented by PSD are in Fig. 1.

There are three threads running in parallel in the sequence diagram: the top thread describes the behaviour of the Redundant Switch; the middle thread describes the behaviour when the driver requests to open the door, and the bottom thread describes the behaviour when the driver requests to close the door. The relationship R : CD − SD is a unidirectional traceability relationship (refer to Sec. 3) that goes from SD to CD, since the objects and terms of SD are dependent on classes, attributes and methods in CD. The traceability between the two models is given implicitly by the SD referencing parts of the CD.

As described in Sec. 2, let us consider a scenario where the system changes, and the redundancy is removed by deleting the Redundant Switch class from the CD. This change represents our slicing criterion given by the megamodel fragment with detail shown in Fig. 5. Note that only the class itself is considered for the impact assessment and not its methods, attributes and associations linked to it.

We now demonstrate the application of the forward megamodel slice algorithm presented in Fig. 4 on the megamodel PSD and the criterion megamodel fragment S_c[PSD].

Line 1 (Initialization): The current slice is initialized to the criterion S_c[PSD] shown as the highlighted parts of Fig. 5.

1st iteration of the outer loop (lines 2-19):

Lines 4-9 (Expansion Level 1): The temporary result S_1[PSD] is initialized to the empty set. Then in lines 5-9, we iterate through the model fragments in the current slice shown in Fig. 5. The CD is considered first and the CD slicer is used. Based on the dependency rule CD1 in Table 1, since the Redundant Switch class is being impacted, all of its attributes and methods are added to the slice and stored in S_1[PSD] on line 8. Since there are no other model fragments to consider on line 5, the loop exits with S_1[PSD] as shown by the highlighted parts in Fig. 6.

Lines 10-17 (Expansion Level 2): Up to this point, R : CD − SD has not been considered in the slicing. In this expansion level, we do use it. First, the level 2 expansion temporary result S_2[PSD] is initialized on line 10 to the empty set. The outer iteration (lines 11-17) is over the model fragments from the level 1 expansion. We first consider the CD. On the opposite end of R : CD − SD is the PowerSlidingDoor : SD (which is M' in the algorithm on line 13). On line 14, we trace through R : CD − SD and add to $S_2[M']$ all the atoms related to those highlighted in the CD. This includes the Redundant Switch object and lifeline and all messages (or parts of them) that are traced back to attributes/methods of the Redundant Switch class in the CD. The result is added to S_2[PSD] on line 15 and can be seen in the highlighted parts of the SD in Fig. 7. Since no other model fragments exist in S_1[PSD] on line 11, the loop exits.

Line 18: The combined result S[PSD] is computed by computing the union of the results of the level 1 and level 2 slices, and can be seen as the result of the 1st iteration of the algorithm in the highlighted parts of Fig. 7.

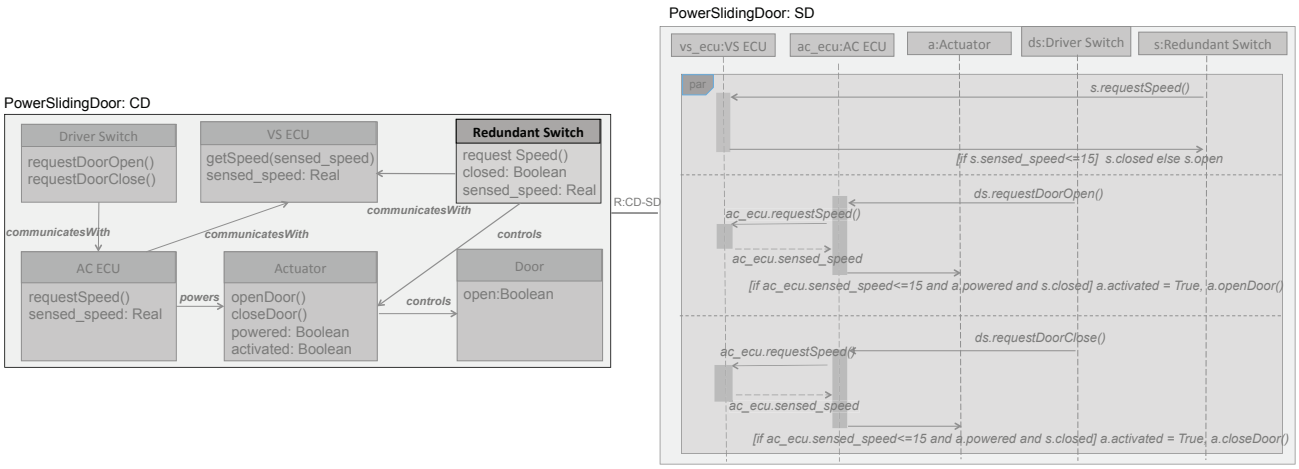

Figure 5: Slicing criterion S_c[PSD].

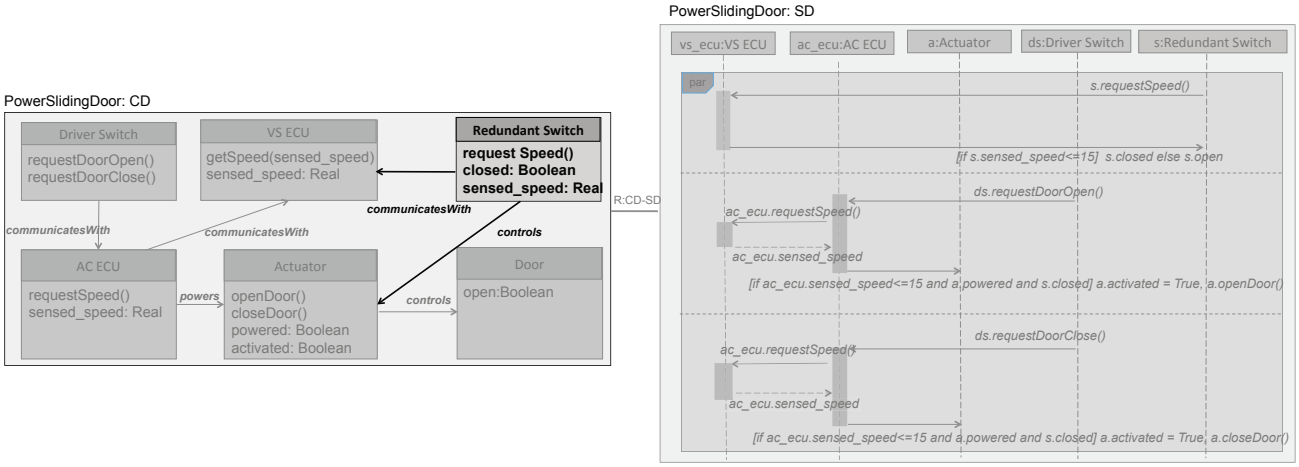

Figure 6: Result of level 1 slicing in 1st iteration.

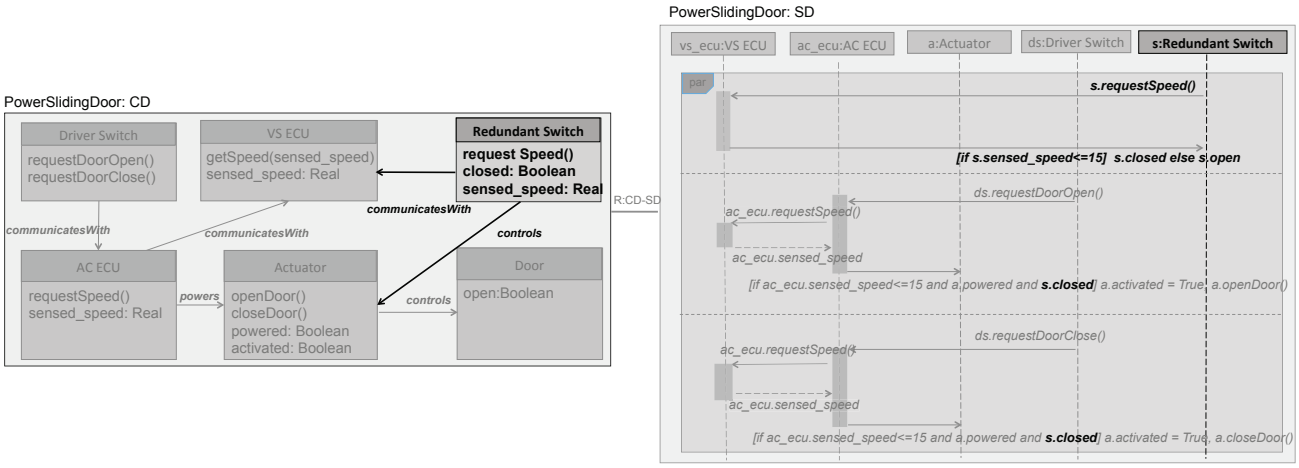

Figure 7: Result of the 1st iteration.

Line 19: In this line, we check to see if any actual expansion has occurred. Since the condition is not met (i.e., the result of the 1st iteration did indeed expand on the initial criterion) we iterate one more time.

2nd iteration of the outer loop (lines 2-19):

The slicing criterion S[PSD] in this iteration is the result of the previous iteration shown in Fig. 7. S_1[PSD] is reset again to the empty set.

Lines 4-9 (Expansion Level 1): First the CD is selected on line 5. Since none of the slicing dependency rules given in Table 1 apply, nothing is added to S_1[PSD] on line 8. Next, the SD is selected on line 5. Now, SD1 − SD3 rules for the SD slicer in Table 1 apply, and the SD slice is expanded to include the arrows of the top two messages and the entire expressions (and therefore messages and arrows) that the term s.closed appears in. This is seen in the highlighted parts of the SD portion of Fig. 8[1].

Lines 10-17 (Expansion Level 2): In this level, tracing across the R : CD − SD relationship from the CD to the SD (recall this is a unidirectional traceability relationship), since no new elements are introduced in the CD slice, nothing is traced to them in the SD. The result is an empty set.

Line 18: The results of the level 1 and the level 2 expansions are unioned and are reflected in the highlighted parts of Fig. 8.

Line 19: Since an expansion (w.r.t. the initial slice for this iteration) has occurred, the condition does not hold, and we iterate one more time on the outer loop.

3rd iteration of the outer loop (lines 2-19):

In this iteration, neither the CD nor the SD are expanded in the first level expansion as none of the dependency rules for their respective slicers holds. Similarly, no new elements are added, and therefore going through the trace links does not identify any other elements to be added to the expansion in level 2. The condition on line 19 now holds (no expansion has occurred), and the main loop of the algorithm exits.

Line 20 (Return): The current slice, S[PSD], which is shown in the highlighted parts of Fig. 8, is returned as the final result of the algorithm.

5.3 Post-processing

As suggested in Sec. 4, we perform a post-processing step, we expand the result of slicing algorithm shown in Fig. 8 to ensure the model fragments are well-formed and contextual information for referential integrity is included.

For the CD, the VS ECU and Actuator classes are included since both endpoints of associations communicatesWith and controls are needed for well-formedness.

For the SD, the VS ECU, AC ECU and Actuator objects and their lifelines are included to satisfy the well-formedness constraint of arrows requiring their lifelines. Also, the execution bar on the leftmost lifeline is included, as both of its input and output arrows are included in the result of the slicing.

Finally, all the methods and attributes of the Actuator class, as well as the sensed_speed attribute of the AC ECU class and the AC ECU class itself are added to satisfy the referential integrity condition between the SD and the CD (they are all referenced in the SD).

The detail of the final megamodel fragment produced after the slicing and post-processing is shown in the highlighted parts of Fig. 9. This can now be used to more efficiently complete the model evolution process by focusing only on the model parts impacted by the original deletion of Redundant Switch in the CD.

[1]Due to space limits, we have skipped visualizing the result at each step of the 2nd iteration and have shown the final result of the union only.

6. RELATED WORK

We identify three main categories of related work: work on model evolution, work on megamodeling operators, and finally, work on model slicing. We describe them below.

Model evolution. A survey on supporting the evolution of UML models in model-driven software development is presented in [14]. The scenarios that cause a model to change are discussed; these form the basis for megamodel evolution in our approach. In [22], the authors discuss some of the key problems of evolution in MDE, summarize the key state-of-the-art, and present some new challenges in research in this area. The problem of model evolution with respect to megamodels is stated as a "dependency heterogeneity" challenge. The authors express the need for a sound, precise theory of heterogeneous dependencies between MDE artefacts, as well as compliant and pragmatic tool support, both of which are complimentary to and/or are part of our current work.

Megamodeling operators. A formal approach to megamodeling, called *Mapping-Aware Megamodeling*, is presented in [9]. Our notion of a megamodel is consistent with it. The approach also describes category theory-based operations on the mapping-aware megamodels, but does not address megamodel slicing. In previous work [24], we presented a set of operators (Map, Filter, Reduce) that can be applied at the megamodel level. We are not aware of any other work in the area of applying operators at the megamodel level, and specifically, we have not seen any work addressing slicing of megamodels.

Model Slicing. We divide this area into work on *specific* model slicers, work on *generic* model slicers and work on slicing *multiple* models.

Specific Model Slicers. Numerous approaches have appeared in the literature describing slicers for specific model types. For example, [13] defines context-free model slicing and presents an algorithm for computing slices on UML class models. [18] also considers UML models, namely, class diagrams, individual state machines, and communicating sets of state machines. The approach achieves slicing of these models using model transformations. An approach for slicing state-based models, in particular, EFSM (extended finite state machine) models, is discussed in [16]. Finally, [17] proposes a slicing technique for UML architectural models, and demonstrates the uses of slicing for different purposes such as regression testing and understanding large architectures. Many other approaches (e.g., [21], [18]) are presented in the literature and can all be used as part of our framework as specific

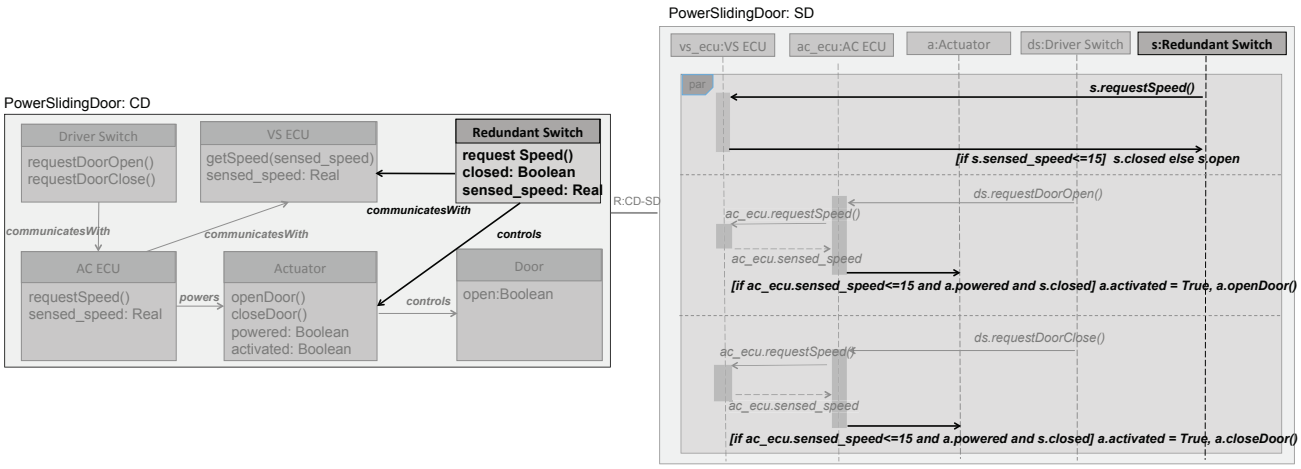

Figure 8: Result of 2nd iteration.

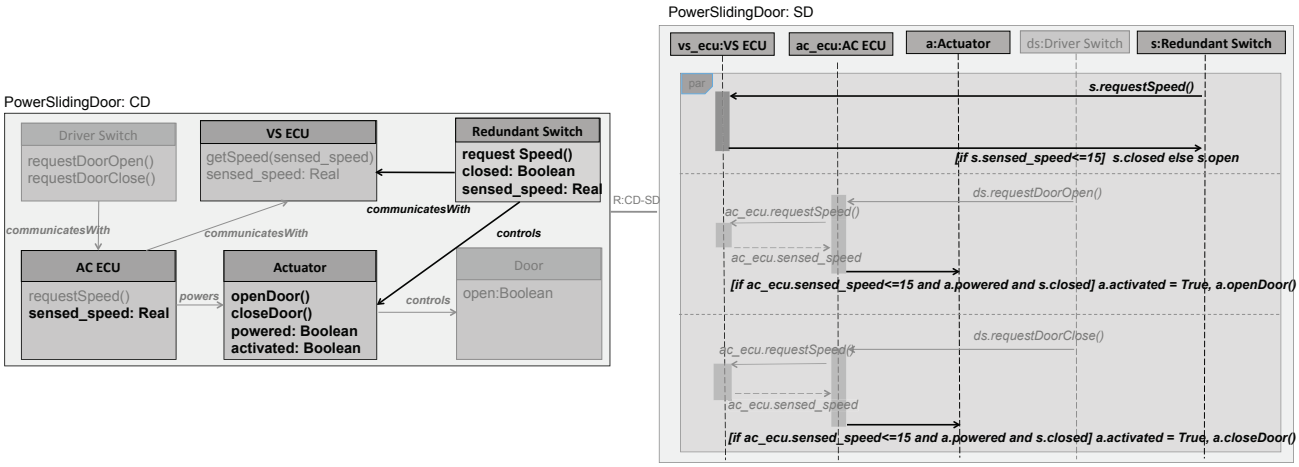

Figure 9: Output of algorithm after post-processing.

model type slicers for each of the model types in our heterogeneous megamodels.

Generic Model Slicers. Generic model slicing has also been studied in the MDE community. For example, the major contribution of [4, 5] is the Kompren language, which provides a generic approach to define a model slicer for a any domain-specific metamodel. The approach permits developers to either use "strict slicers" that output models which conform to their expected metamodel, or to define "soft slicers" that can output nonconforming models or even outputs that are not models. Although Kompren can be used for identifying specific type slicers in our framework, it is not applicable for megamodel slicing, where a megamodel slicer has to carefully invoke the specific type slicers. The work in [7] defines slicing at a theoretical level, whereas we focus on a more pragmatic approach. Also, the same work focuses on dynamic slicing, as does the transformation slicing work in [25], whereas our approach is considered a static slicing approach. As far as we know, none of the approaches in this category directly address megamodel slicing (whether the megamodels are heterogeneous or not).

Slicing Multiple Models. Although the work presented in [7] does not primarily focus on megamodel slicing, it briefly discusses heterogeneous slicing as the union of individual slicers. A slicing theory is presented at a high level and does not go into the details of implementing a megamodel slicing algorithm. From the modeling and safety community, [20] proposes a batch model slicer for slicing SysML models related to safety requirements. [10] presents a prototype tool called SafeSlice which performs the slicing needed in [20]. This line of work performs slicing on specific model types, whereas our work is a generic slicing approach. Also, the presented approach is amorphous slicing, where the result of the slice is not a model fragment of the original system. For example, transitions are added to sliced state-machines in order to preserve their behaviour. Our current approach only considers slices to be fragments of the original model (non-amorphous); however, we do plan to look at amorphous slicing in future work.

7. CONCLUSION

Model slicing is a useful technique for assessing change impact during model evolution activities. Although slicing of individual models has been investigated, slicing of heterogeneous model collections has received much less attention. In this paper, we have proposed a general algorithm for

slicing of heterogeneous model collections represented using megamodels and illustrated the algorithm on an automotive example. We analyzed the algorithm and showed that it behaves as expected with respect to termination, correctness, time complexity and minimality. Finally, we discussed the issues concerning slice well-formedness and referential integrity as well as how to generalize the algorithm to support arbitrary relationship types, N-ary relationship and nested megamodels. We are currently developing tooling for the algorithm using the Model Management INTeractive (MMINT) framework [8] and plan to use it to conduct more extensive case studies to better understand the strengths and weaknesses of the approach.

8. ACKNOWLEDGMENTS

This work is being done as part of the NECSIS project (www.necsis.ca), funded by Automotive Partnership Canada and NSERC.

9. REFERENCES

[1] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3):515–526, 2006.

[2] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Proc. of CIDR'03*, volume 2003, pages 209–220, 2003.

[3] J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. In *Proc. of OOPSLA/GPCE Workshops*, 2004.

[4] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Modeling Model Slicers. In *Proc. of MODELS'11*, pages 62–76. Springer, 2011.

[5] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Kompren: Modeling and Generating Model Slicers. *SoSyM*, 14(1):321–337, 2015.

[6] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A Manifesto for Model Merging. In *Proc. of GAMMA@ICSE'06*, pages 5–12. ACM, 2006.

[7] T. Clark. A General Model-Based Slicing Framework. In *Proc. of Wrksp on Composition and Evolution of Model Transformations*, 2011.

[8] A. Di Sandro, R. Salay, M. Famelis, S. Kokaly, and M. Chechik. MMINT: A Graphical Tool for Interactive Model Management. In *Proc. of MODELS'15 (demo track)*, 2015.

[9] Z. Diskin, S. Kokaly, and T. Maibaum. Mapping-Aware Megamodeling: Design Patterns and Laws. In *Proc. of SLE'13*, pages 322–343, 2013.

[10] D. Falessi, S. Nejati, M. Sabetzadeh, L. Briand, and A. Messina. SafeSlice: A Model Slicing and Design Safety Inspection Tool for SysML. In *Proc. of ESEC/FSE'11*, pages 460–463. ACM, 2011.

[11] O. Gotel and A. Finkelstein. Contribution structures. In *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on*, pages 100–107. IEEE, 1995.

[12] International Organization for Standardization. *ISO 26262: Road Vehicles – Functional Safety*, 2011. 1st version.

[13] H. Kagdi, J. I. Maletic, and A. Sutton. Context-Free Slicing of UML Class Models. In *Proc. of ICSM'05*, pages 635–638. IEEE, 2005.

[14] A. Khalil and J. Dingel. Supporting the Evolution of UML Models in Model Driven Software Development: a Survey. Technical Report 602, School of Computing, Queen's University, Ontario, Canada, 2013.

[15] S. Kokaly, R. Salay, V. Cassano, T. Maibaum, and M. Chechik. A Model Management Approach for Assurance Case Reuse due to System Evolution. In *Proc. of MODELS'16*, 2016. (to appear).

[16] B. Korel, I. Singh, L. Tahat, and B. Vaysburg. Slicing of State-Based Models. In *Proc. of ICSM'03*, pages 34–43. IEEE, 2003.

[17] J. T. Lallchandani and R. Mall. A Dynamic Slicing Technique for UML Architectural Models. *IEEE TSE*, 37(6):737–771, 2011.

[18] K. Lano and S. Kolahdouz-Rahimi. Slicing of UML Models Using Model Transformations. In *Proc. of MODELS'10*, pages 228–242. Springer, 2010.

[19] B. Li, X. Sun, H. Leung, and S. Zhang. A Survey of Code-Based Change Impact Analysis Techniques. *J. Software Testing, Verification and Reliability*, 23(8):613–646, 2013.

[20] S. Nejati, M. Sabetzadeh, D. Falessi, L. Briand, and T. Coq. A SysML-based Approach to Traceability Management and Design Slicing in Support of Safety Certification: Framework, Tool Support, and Case Studies. *Information and Software Technology*, 54(6):569–590, 2012.

[21] K. Noda, T. Kobayashi, K. Agusa, and S. Yamamoto. Sequence Diagram Slicing. In *Proc. of APSEC'09*, pages 291–298. IEEE, 2009.

[22] R. F. Paige, N. Matragkas, and L. M. Rose. Evolving Models in Model-Driven Engineering: State-of-the-art and Future Challenges. *J. of Systems and Software*, 111:272–280, 2016.

[23] R. Salay, M. Famelis, J. Rubin, A. Di Sandro, and M. Chechik. Lifting Model Transformations to Product Lines. In *Proc. of ICSE'14*, pages 117–128. ACM, 2014.

[24] R. Salay, S. Kokaly, A. Di Sandro, and M. Chechik. Enriching Megamodel Management with Collection-Based Operators. In *Proc. of MODELS'15*, pages 236–245, 2015.

[25] Z. Ujhelyi, Á. Horváth, and D. Varró. Towards dynamic backward slicing of model transformations. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 404–407. IEEE, 2011.

[26] M. Weiser. Program Slicing. In *Proc. of ICSE'81*, pages 439–449. IEEE Press, 1981.