A Formal Framework for Incremental Model Slicing

Gabriele Taentzer[1], Timo Kehrer[2], Christopher Pietsch[3](✉), and Udo Kelter[3]

[1] Philipps-Universität Marburg, Marburg, Germany
[2] Humboldt-Universität zu Berlin, Berlin, Germany
[3] University of Siegen, Siegen, Germany
cpietsch@informatik.uni-siegen.de

Abstract. Program slicing is a technique which can determine the simplest program possible that maintains the meaning of the original program w.r.t. a slicing criterion. The concept of slicing has been transferred to models, in particular to statecharts. In addition to the classical use cases of slicing adopted from the field of program understanding, model slicing is also motivated by specifying submodels of interest to be further processed more efficiently, thus dealing with scalability issues when working with very large models. Slices are often updated throughout specific software development tasks. Such a slice update can be performed by creating the new slice from scratch or by incrementally updating the existing slice. In this paper, we present a formal framework for defining model slicers that support incremental slice updates. This framework abstracts from the behavior of concrete slicers as well as from the concrete model modification approach. It forms a guideline for defining incremental model slicers independent of the underlying slicer's semantics. Incremental slice updates are shown to be equivalent to non-incremental ones. Furthermore, we present a framework instantiation based on the concept of edit scripts defining application sequences of model transformation rules. We implemented two concrete model slicers for this instantiation based on the Eclipse Modeling Framework.

1 Introduction

Program slicing as introduced by Weiser [1] is a technique which determines those parts of a program (the *slice*) which may affect the values of a set of (user-)selected variables at a specific point (the *slicing criterion*). Since the seminal work of Weiser, which calculates a slice by utilizing static data and control flow analysis and which primarily focuses on assisting developers in debugging, a plethora of program slicing techniques addressing a broad range of use cases have been proposed [2].

With the advent of Model-Driven Engineering (MDE) [3], models rather than source code play the role of primary software development artifacts. Similar use cases as known from program slicing must be supported for model slicing [4–6]. In addition to classical use cases adopted from the field of program understanding, model slicing is often motivated by scalability issues when working with very large models [7,8], which has often been mentioned as one of the biggest obstacles in applying MDE in practice [9,10]. Modeling frameworks such as the Eclipse Modeling Framework (EMF) and widely-used model management tools do not scale beyond a few tens of thousands of model elements [11], while large-scale industrial models are considerably larger [12]. As a consequence, such models cannot even be edited in standard model editors. Thus, the *extraction of editable submodels from a larger model* is the only viable solution to support an efficient yet independent editing of huge monolithic models [8]. Further example scenarios in which model slices may be constructed for the sake of efficiency include model checkers, test suite generators, etc., in order to reduce runtimes and memory consumption.

Slice criteria are often modified during software development tasks. This leads to corresponding *slice updates* (also called slice *adaptations* in [8]). During a debugging session, e.g., the slicing criterion might need to be modified in order to closer inspect different debugging hypotheses. The independent editing of submodels is another example of this. Here, a slice created for an initial slicing criterion can turn out to be inappropriate, most typically because additional model elements are desired or because the slice is still too large. These *slice update* scenarios have in common that the original slicing criterion is modified and that the existing slice must be updated w.r.t. the new slicing criterion.

Model slicing is faced with two challenging requirements which do not exist or which are of minor importance for traditional program slicers. First, the increasing importance and prevalence of domain-specific modeling languages (DSMLs) as well as a considerable number of different use cases lead to a huge number of different concrete slicers, examples will be presented in Sect. 2. Thus, methods for developing model slicers should abstract from a slicer's concrete behavior (and thus from concrete modeling languages) as far as possible. Ideally, model slicers should be generic in the sense that the behavior of a slicer is *adaptable* with moderate configuration effort [7]. Second, rather than creating a new slice from scratch for a modified slicing criterion, slices must often be updated *incrementally*. This is indispensable for all use cases where slices are edited by developers since otherwise these slice edits would be blindly overwritten [8]. In addition, incremental slice updating is a desirable feature when it is more efficient than creating the slice from scratch. To date, both requirements have been insufficiently addressed in the literature.

In this paper, we present a fundamental methodology for developing model slicers which abstract from the behavior of a concrete slicer and which support incremental model slicing. To be independent of a concrete DSML and use cases, we restrict ourselves to static slicing in order to support both executable and non-executable models. We make the following contributions:

1. A formal framework for incremental model slicing which can function as a guideline for defining adaptable and incremental model slicers (s. Sect. 3).

This framework is based on graph-based models and model modifications and abstracts from the behavior of concrete slicers as well as from the concrete model modification approach. Within this framework we show that incremental slice updates are equivalent to non-incremental ones.

2. An instantiation of this formal framework where incremental model slicers are specified by model patches. Two concrete model slicers.

2 Motivating Example

In this section we introduce a running example to illustrate two use cases of model slicing and to motivate incremental slice updates.

Figure 1 shows an excerpt of the system model of the *Barbados Car Crash Crisis Management System (bCMS)* [13]. It describes the operations of a police and a fire department in case of a crisis situation.

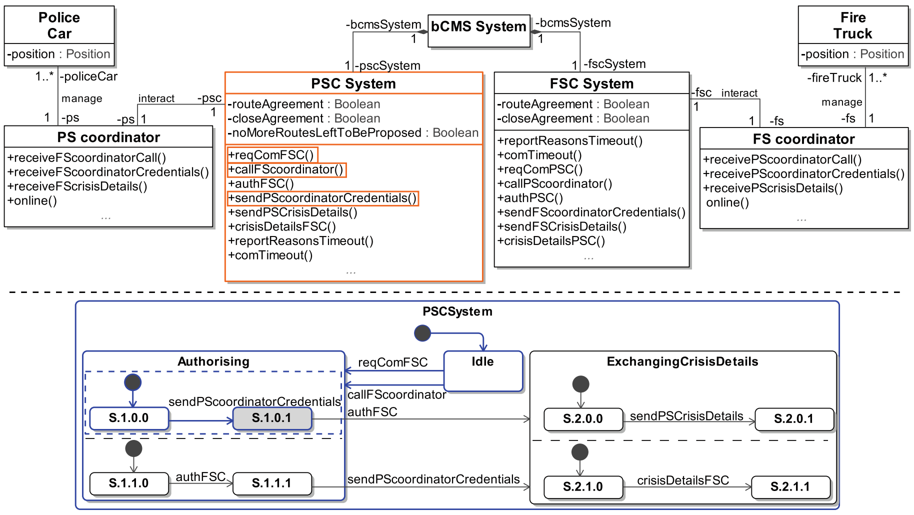

Fig. 1. Excerpt of the system model of the bCMS case study [13].

The system is modeled from different viewpoints. The class diagram models the key entities and their relationships from a static point of view. A police station coordinator (`PS coordinator`) and a fire station coordinator (`FS coordinator`) are responsible for coordinating and synchronizing the activities on the police and fire station during a crisis. The interaction of both coordinators is managed by the respective system classes `PSC System` and `FSC System` which contain several operations for, e.g., establishing the communication between the coordinators and exchanging crisis details. The state machine diagram models the dynamic view of the class `PSC System`, i.e., its runtime behavior, for sending and receiving authorization credentials and crisis details to and from a `FSC System`. Initially, the `PSC System` is in the state `Idle`. The establishment of the communication can be triggered by calling the operation `callFScoordinator` or `reqComFSC`. In the composite state `Authorising` the system waits for exchanging the credentials of the `PS` and `FS coordinator` by calling the operation `sendPScoordinatorCredentials` and `authFSC`, or vice versa. On entering the composite state `ExchangingCrisisDetails`, details can be sent by the operation call `sendPSCrisisDetails` or details can be received by the operation call `crisisDetailsFSC`.

Model Slicing. Model slicers are used to find parts of interest in a given model M. These parts of M are specified by a *slicing criterion*, which is basically a set of model elements or, more formally, a submodel C of M. A slicer extends C with further model elements of M according to the purpose of the slicer.

We illustrate this with two use cases. Use case **A** is known as *backward slicing* in state-based models [4]. Given a set of states C in a statechart M as slicing criterion, the slicer determines all model elements which may have an effect on states in C. For instance, using `S.1.0.1` (s. gray state in Fig. 1) as slicing criterion, the slicer recursively determines all incoming transitions and their sources, e.g., the transition with the event `sendPScoordinatorCredentials` and its source state `S.1.0.0`, until an initial state is reached.

The complete backward slice is indicated by the blue elements in the lower part of Fig. 1. The example shows that our general notion of a slicing criterion may be restricted by concrete model slicers. In this use case, the slicing criterion must not be an arbitrary submodel of a given larger model, but a very specific one, i.e., a set of states.

Use case **B** is the *extraction of editable models* as presented in [8]. Here, the slicing criterion C is given by a set of *requested model elements* of M. The purpose of this slicer is to find a submodel which is editable and which includes all requested model elements. For example, if we use the blue elements in the lower part of Fig. 1 as slicing criterion, the model slice also contains the orange elements in the upper part of Fig. 1, namely three operations, because events of a transitions in a statechart represent operations in the class diagram, and the class containing these operations.

Slice Update. The slicing criterion might be updated during a development task in order to obtain an updated slice. It is often desirable to update the slice rather than creating the new slice from scratch, e.g., because this is more efficient. Let us assume in use case **A** that the slicing criterion changes from `S.1.0.1` to `S.1.1.1`. The resulting model slice only differs in the contained regions of the composite state `Authorising`. The upper region and its contained elements would be removed, while the lower region and its contained elements would be added. Next we could use the updated model slice from use case **A** as slicing criterion in use case **B**. In the related resulting model slice, the operation `sendPScoordinatorCredentials` would then be replaced by the operation `authFSC`.

3 Formal Framework

We have seen in the motivating example that model slicers can differ considerably in their intended purpose. The formal framework we present in the following defines the fundamental concepts for model slicing and slice updates. This framework uses graph-based models and model modifications [14]. It shall serve as a guideline how to define model slicers that support incremental slice updates.

3.1 Models as Graphs

Considering models, especially visual models, their concrete syntax is distinguished from their abstract one. In Fig. 1, a UML model is shown in its concrete representation. In the following, we will reason about their underlying structure, i.e., their abstract syntax, which can be considered as graph. The abstract syntax of a modeling language is usually defined by a meta-model which contains the type information about nodes and edges as well as additional constraints. We assume that a meta-model is formalized by an attributed graph; model graphs are defined as attributed graphs being typed over the meta-model. This typing can be characterized by an attributed graph morphism [15]. In addition, graph constraints [16] may be used to specify additional requirements. Due to space limitations, we do not formalize constraints in this paper.

Definition 1 (Typed model graph and morphism). *Given two attributed graphs M and MM, called* model *and* meta-model, *the typed model (graph) of M is defined as $M^T = (M, type^M)$ with $type^M : M \to MM$ being an attributed graph morphism, called* typing morphism[1]. *Given two typed models M and N, an attributed graph morphism $f : M \to N$ is called* typed model morphism *if $type^N \circ f = type^M$.*

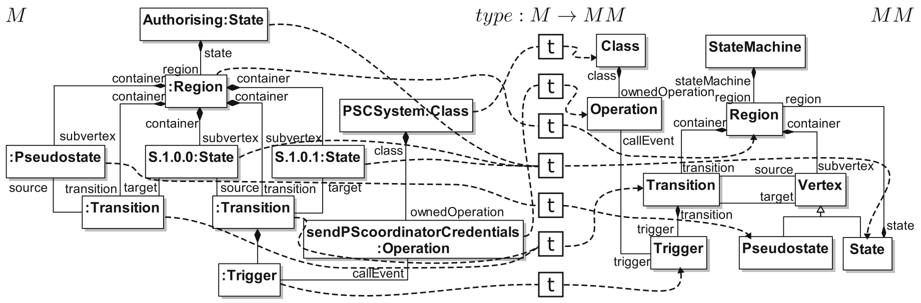

Fig. 2. Excerpt of a typed model graph.

Example 1 (Typed model graph). The left-hand side of Fig. 2 shows the model graph of an excerpt from the model depicted in Fig. 1. The model graph is

[1] In the following, we usually omit the adjective "attributed".

typed over the meta-model depicted on the right-hand side of Fig. 2. It shows a simplified excerpt of the UML meta-model. Every node (and edge) of the model graph is mapped onto a node or edge of the type graph by the graph morphism $\mathit{type} : M \to MM$.

Typed models and morphisms as defined above form the category $\mathbf{AGraphs}_{ATG}$ in [15]. It has various properties since it is an adhesive HLR category using a class $\mathcal{M}$ of injective graph morphisms with isomorphic data mapping, it has pushouts and pullbacks where at least one morphism is in $\mathcal{M}$. These constructions can be considered as generalized union and intersection of models being defined component-wise on nodes and edges such that they are structure-compatible. These constructions are used to define the formal framework.

3.2 Model Modifications

If we do not want to go into any details of model transformation approaches, the temporal change of models is roughly specified by model modifications. Each model modification describes the original model, an intermediate one after having performed all intended element deletions, and the resulting model after having performed all element additions.

Definition 2 (Model modification). *Given two models M_1 and M_2, a (direct)* model modification $M_1 \Longrightarrow M_2$ *is a span of injective morphisms* $M_1 \xleftarrow{m_1} M_s \xrightarrow{m_2} M_2$.

1. *Two model modifications* $M_1 \xleftarrow{m_{11}} M_{12} \xrightarrow{m_{12}} M_2$ *and* $M_2 \xleftarrow{m_{22}} M_{23} \xrightarrow{m_{23}} M_3$ *are* concatenated *to model modification* $M_1 \xleftarrow{m_{13}} M_{13} \xrightarrow{m_{33}} M_3$ *with* (m_{13}, m_{33}) *being the pullback of* m_{12} *and* m_{22} *(intersecting* M_{12} *and* M_{23}*).*
2. *Given two direct model modifications* $m : M_1 \xleftarrow{m_1} M_s \xrightarrow{m_2} M_2$ *and* $p : P_1 \xleftarrow{p_1} P_s \xrightarrow{p_2} P_2$, *$p$ can be* embedded *into m, written* $e : p \to m$, *if there are injective morphisms (also called embeddings)* $e_1 : P_1 \to M_1$, $e_s : P_s \to M_s$, *and* $e_2 : P_2 \to M_2$ *with* $e_1 \circ p_1 = m_1 \circ e_s$ *and* $e_2 \circ p_2 = m_2 \circ e_s$.
3. *A sequence* $M_0 \Longrightarrow M_1 \Longrightarrow \ldots \Longrightarrow M_n$ *of direct model modifications is called* model modification *and is denoted by* $M_0 \overset{*}{\Longrightarrow} M_n$.
4. *There are five special kinds of model modifications:*
   (a) *Model modification* $M \xleftarrow{id_M} M \xrightarrow{id_M} M$ *is called* identical.
   (b) *Model modification* $\emptyset \longleftarrow \emptyset \longrightarrow \emptyset$ *is called* empty.
   (c) *Model modification* $\emptyset \longleftarrow \emptyset \longrightarrow M$ *is called* model creation.
   (d) *Model modification* $M \longleftarrow \emptyset \longrightarrow \emptyset$ *is called* model deletion.
   (e) $M_2 \xleftarrow{m_2} M_s \xrightarrow{m_1} M_1$ *is called* inverse modification *to* $M_1 \xleftarrow{m_1} M_s \xrightarrow{m_2} M_2$.

In a direct model modification, model M_s characterizes an intermediate model where all deletion actions have been performed but nothing has been added yet. To this end, M_s is the intersection of M_1 and M_2.

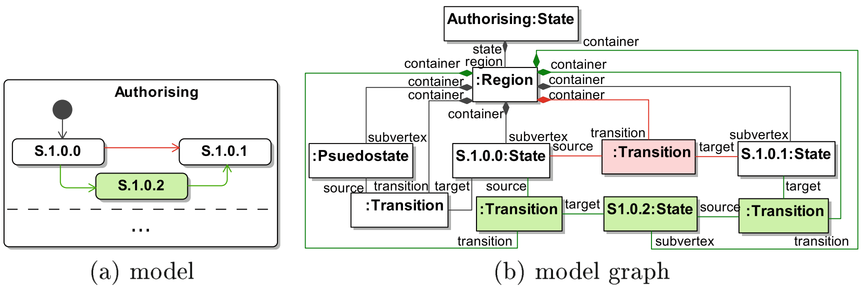

Fig. 3. Excerpt of a model modification

Example 2 (Direct model modification). Figure 3 shows a model modification using our running example. While Fig. 3(a) focuses on the concrete model syntax, Fig. 3(b) shows the changing abstract syntax graph. Figure 3(a) depicts an excerpt of the composite state `Authorising`. The red transition is deleted while the green state and transitions are created. The model modification $m : M_1 \xleftarrow{m_1} M_s \xrightarrow{m_2} M_2$ is illustrated in Fig. 3(b). The red elements represent the set of nodes (and edges) $M_1 \setminus m_1(M_s)$ to be deleted. The set $M_2 \setminus m_2(M_s)$ describing the nodes (and edges) to be created is illustrated by the green elements. All other nodes (and edges) represent the intermediate model M_s.

The *double pushout* approach to graph transformation [15] is a special kind of model modification:

Definition 3 (Rule application). *Given a model G and a model modification $r : L \xleftarrow{l} K \xrightarrow{r} R$, called* rule, *with injective morphism $m : L \to G$, called* match, *the rule application $G \Longrightarrow_{r,m} H$ is defined by the following two pushouts:*

$$\begin{tikzcd} L \arrow[d, "m"'] \arrow[dr, phantom, "(PO1)"] & K \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(PO2)"] & R \arrow[d, "m'"] \\ G & D \arrow[l] \arrow[r] & H \end{tikzcd}$$

Model H is constructed in two passes: (1) $D := G \setminus m(L \setminus l(K))$, i.e., erase all model elements that are to be deleted; (2) $H := D \cup m'(R \setminus r(K))$ such that a new copy of all model elements that are to be created is added.

Note that the first pushout above exists if $G \setminus m(L \setminus l(K))$ does not yield dangling edges [15]. It is obvious that the result of a rule application $G \Longrightarrow_r H$ is a direct model modification $G \xleftarrow{g} D \xrightarrow{h} H$.

3.3 Model Slicing

In general, a model slice is an interesting part of a model comprising a given slicing criterion. It is up to a concrete slicing definition to specify which model parts are of interest.

Definition 4 (Model slice). *Given a model M and a* slicing criterion C *with a morphism $c : C \to M$. A* model slice $S = \mathit{Slice}(M, c)$ *is a model S such that there are two morphisms $m : S \to M$ and $e : C \to S$ with $m \circ e = c$.*

Note that each model slice $S = \mathit{Slice}(M, c)$ induces a model modification $C \xleftarrow{id_C} C \xrightarrow{e} S$.

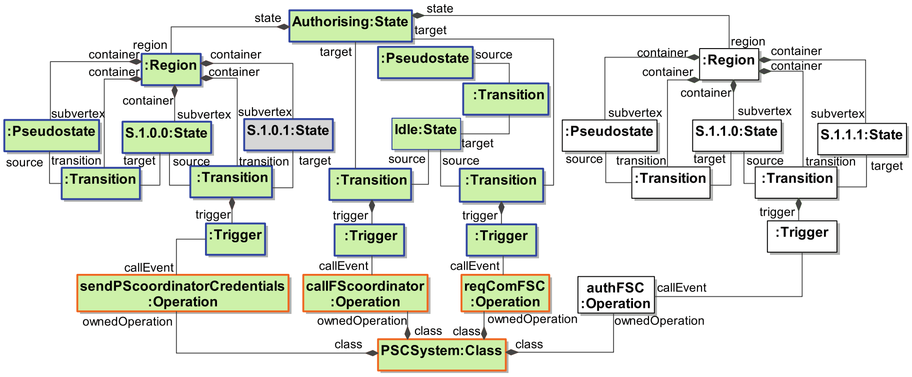

Fig. 4. Excerpt of two model slices

Example 3 (Model slice). Figure 4 depicts an excerpt of the model graph of M depicted in Fig. 1 and the two slices $S_{back} = \mathit{Slice}(M, c_{back})$ and $S_{edit} = \mathit{Slice}(M, c_{edit})$. S_{back} is the backward slice as informally described in Sect. 2. $C_{back} = \{\texttt{S.1.0.1}\}$ is the first slice criterion. The embedding $c_{back}(C_{back})$ is represented by the gray-filled element while embedding $m_{back}(S_{back})$ is represented by the blue-bordered elements. Model $e_{back}(C_{back})$ is illustrated by the gray-filled state having a blue border and $S_{back} \setminus e_{back}(C_{back})$ by the green-filled elements having a blue border.

Let S_{back} be the slicing criterion for the slice S_{edit}, i.e. $C_{edit} = S_{back}$ and $c_{edit}(C_{edit}) = m_{back}(S_{back})$. S_{edit} is the extracted editable submodel introduced in Sect. 2 by use case **B**. Its embedding $m_{edit}(S_{edit})$ is represented by the blue and orange-bordered elements. Model $e_{edit}(C_{edit})$ is illustrated by the blue-bordered elements and $S_{edit} \setminus e_{edit}(C_{edit})$ by the green-filled elements having an orange border.

3.4 Incremental Slice Update

Throughout working with a model slice, it might happen that the slice criterion has to be modified. The update of the corresponding model slice can be performed incrementally. Actually, modifying slice criteria can happen rather frequently in practice by, e.g., editing independent submodels of a large model in cooperative work.

Definition 5 (Slice update construction). *Given a model slice $S_1 = Slice(M, C_1 \to M)$ and a direct model modification $c = C_1 \xleftarrow{c_1} C_s \xrightarrow{c_2} C_2$, slice $S_2 = Slice(M, C_2 \to M)$ can be constructed as follows:*

1. *Given slice S_1 we deduce the model modification $C_1 \xleftarrow{id_{C_1}} C_1 \xrightarrow{e_1} S_1$ and take its inverse modification: $S_1 \xleftarrow{e_1} C_1 \xrightarrow{id_{C_1}} C_1$.*
2. *Then we take the given model modification c for the slice criterion.*
3. *And finally we take the model modification $C_2 \xleftarrow{id_{C_2}} C_2 \xrightarrow{e_2} S_2$ induced by slice S_2.*

All model modifications are concatenated yielding the direct model modification $S_1 \xleftarrow{e_1 \circ c_1} C_s \xrightarrow{e_2 \circ c_2} S_2$ called slice update construction *(see also Fig. 6).*

Example 4 (Slice update example). Figure 5 illustrates a slice update construction with $S_{edit} = Slice(M, C_{edit} \to M)$ being the extracted submodel of our previous example illustrated by the red-dashed box. The modification $c : C_{edit} \xleftarrow{c_{edit}} C_s \xrightarrow{c_{edit'}} C_{edit'}$ of the slicing criterion is depicted by the gray-filled elements. The red-bordered elements represent the set $C_s \backslash c_{edit}(C_{edit})$ of elements removed from the slicing criterion. The green-bordered elements form the set $C_s \setminus c_{edit'}(C_{edit'})$ of elements added to the slicing criterion. $S_{edit'} = Slice(M, C_{edit'} \to M)$ is the extracted submodel represented by the green-dashed box. Consequently, the slice is updated by deleting all elements in $S_{edit} \setminus e_{edit}(c_{edit}(C_s))$, represented by the red-bordered and red- and white-filled elements, and adding all elements in $S_{edit'} \backslash e_{edit'}(c_{edit'}(C_s))$, represented by the green-bordered and green- and white-filled elements. Note that the white-filled elements are removed and added again. This motivated us to consider incremental slice updates defined below.

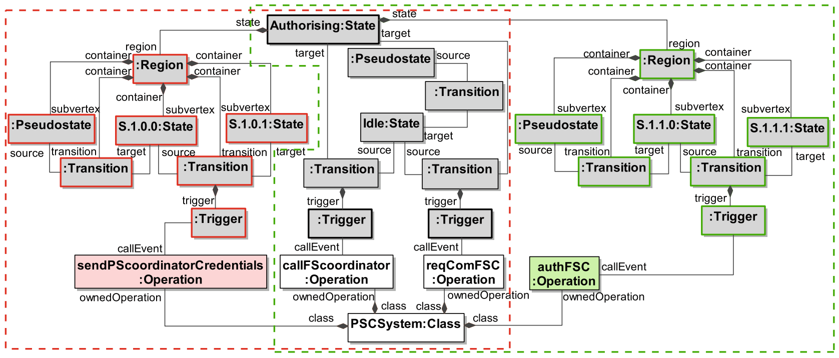

Fig. 5. Excerpt of an (incremental) slice update.

Definition 6 (Incremental slice update). *Given M and $C_1 \to M_1$ as in Definition 4 as well as a direct model modification $C_1 \xleftarrow{c_1} C_s \xrightarrow{c_2} C_2$, model slice $S_1 = Slice(M, C_1 \to M)$ is incrementally updated to model slice $S_2 = Slice(M, C_2 \to M)$ yielding a direct model modification $S_1 \xleftarrow{s_1} S_s \xrightarrow{s_2} S_2$, called* incremental slice update *from S_1 to S_2, with s_1 and s_2 being the pullback of $m_1 : S_1 \to M$ and $m_2 : S_2 \to M$ (see also Fig. 6).*

Example 5 (Incremental slice update example). Given S_{edit} and $S_{edit'}$ of our previous example. Furthermore, given the model modification $S_{edit} \xleftarrow{s_{edit}} S_s \xrightarrow{s_{edit'}} S_{edit'}$ whereby S_s is isomorphic to the intersection of S_{edit} and $S_{edit'}$ in M, i.e. $m_s : S_s \to m_{edit}(S_{edit}) \cap m_{edit'}(S_{edit'})$ with m_s being an isomorphism due to the pullback construction. S_s is illustrated by the elements contained in the intersection of the red- and green-dashed box in Fig. 5. In contrast to the slice update construction of the previous example the white-filled elements are not affected by the incremental slice update.

$$\begin{tikzcd}
C_1 \arrow[d, "e_1"'] & C_s \arrow[l, "c_1"'] \arrow[r, "c_2"] \arrow[d, dashed, "e_s"'] & C_2 \arrow[d, "e_2"] \\
S_1 \arrow[dr, "m_1"'] & S_s \arrow[l, "s_1"'] \arrow[r, "s_2"] \arrow[d, dashed, "m_s"'] & S_2 \arrow[dl, "m_2"] \\
 & M &
\end{tikzcd}$$

Fig. 6. Incremental slice update

Ideally, the slice update construction in Definition 5 should not yield a different update than the incremental one. However, this is not the case in general since the incremental update keeps as many model elements as possible in contrast to the update construction in Definition 5 In any case, both update constructions should be compatible with each other, i.e., should be in an embedding relation, as stated on the following proposition.

Proposition 1 (Compatibility of slice update constructions). *Given M and C_1 as in Definition 4 as well as a direct model modification $C_1 \xleftarrow{c_1} C_s \xrightarrow{c_2} C_2$, the model modification resulting from the slice update construction in Definition 5 can be embedded into the incremental slice update from S_1 to S_2 (see also Fig. 6).*

Proof idea: Given an incremental slice update $S_1 \xleftarrow{s_1} S_s \xrightarrow{s_2} S_2$, it is the pullback of $m_1 : S_1 \to M$ and $m_2 : S_2 \to M$. The slice update construction yields $m_1 \circ e_1 \circ c_1 = m_2 \circ e_2 \circ c_2$. Due to pullback properties there is a unique embedding $e : C_s \to S_s$ with $s_1 \circ e = e_1 \circ c_1$ and $s_2 \circ e = e_2 \circ c_2$.[2]

4 Instantiation of the Formal Framework

In this section, we present an instantiation of our formal framework which is inspired by the model slicing tool introduced in [8]. The basic idea of the approach is to create and incrementally update model slices by calculating and applying a special form of model patches, introduced and referred to as edit script in [17].

[2] This proof idea can be elaborated to a full proof in a straight forward manner.

4.1 Edit Scripts as Refinements of Model Modifications

An *edit script* $\Delta_{M_1 \Rightarrow M_2}$ specifies how to transform a model M_1 into a model M_2 in a stepwise manner. Technically, this is a data structure which comprises a set of rule applications, partially ordered by an acyclic dependency graph. Its nodes are rule applications and its edges are dependencies between them [17]. Models are represented as typed graphs as in Definition 1, rule applications are defined as in Definition 3. Hence, the semantics of an edit script is a set of rule application sequences taking all possible orderings of rule applications into account. Each sequence can be condensed into the application of one rule following the concurrent rule construction in, e.g., [15]. Hence, an edit script $\Delta_{M_1 \Rightarrow M_2}$ induces a set of model modifications of the form $M_1 \xleftarrow{m_1} M_s \xrightarrow{m_2} M_2$.

Given two models M_1 and M_2 as well as a set R of transformation rules for this type of models, edit scripts are calculated in two basic steps [17]:

First, the corresponding elements in M_1 and M_2 are calculated using a model matcher [18]. A basic requirement is that such a matching can be formally represented as a (partial) injective morphism $c : M_1 \rightarrow M_2$. If so, the matching morphism c yields a unique model modification $m : M_1 \xleftarrow{\supseteq} M_s \xrightarrow{m_2} M_2$ (up to isomorphism) with $m_2 = c|_{M_s}$. This means that M_s always has to be a graph.

Second, an edit script is derived. Elementary model changes can be directly derived from a model matching; elements in M_1 and M_2 which are not involved in a correspondence can be considered as deleted and added, respectively [19]. The approach presented in [17] partitions the set of elementary changes such that each partition represents the application of a transformation rule of the given set R of transformation rules [20], and subsequently calculates the dependencies between these rule applications [17], yielding an edit script $\Delta_{M_1 \Rightarrow M_2}$. Sequences of rule applications of an edit script do not contain transient effects [17], i.e., pairs of change actions which cancel out each other (such as creating and later deleting one and the same element). Thus, no change actions are factored out by an edit script.

4.2 Model Slicing Through Slice-Creating Edit Scripts

Edit scripts are also used to construct new model slices. Given a model M and a slicing criterion C, a *slice-creating edit script* $\Delta_{\epsilon \Rightarrow S}$ is calculated which, when applied to the empty model ϵ, yields the resulting slice S. The basic idea to construct $\Delta_{\epsilon \Rightarrow S}$ is to consider the model M as created by an edit script $\Delta_{\epsilon \Rightarrow M}$ applied to the empty model ϵ and to identify a sub-script of $\Delta_{\epsilon \Rightarrow M}$ which (at least) creates all elements of C. The slice creating edit script $\Delta_{\epsilon \Rightarrow S}$ consists of the subgraph of the dependency graph of the model-creating edit script $\Delta_{\epsilon \Rightarrow M}$ containing (i) all nodes which create at least one model element in C, and (ii) all required nodes and connecting edges according to the transitive closure of the "required" relation, which is implied by dependencies between rule applications.

Since the construction of edit scripts depends on a given set R of transformation rules, *a basic applicability condition is that all possible models and all possible slices can be created by rules available in R.* Given that this condition is satisfied, model slicing through slice-creating edit scripts indeed behaves according to Definition 4, i.e., a slice $S = Slice(M, C \rightarrow M)$ is obtained by applying $\Delta_{\epsilon \Rightarrow S}$ to the empty model: The resulting slice S is a submodel of M and a supermodel of C. As we will see in Sect. 5, the behavior of a concrete model slicer and thus its intended purpose is configured by the transformation rule set R.

4.3 Incremental Slicing Through Slice-Updating Edit Scripts

To incrementally update a slice $S_1 = Slice(M, C_1 \rightarrow M)$ to become slice $S_2 = Slice(M, C_2 \rightarrow M)$, we show that the approach presented in [8] constructs a *slice-updating edit script* $\Delta_{S_1 \Rightarrow S_2}$ which, if applied to the current slice S_1, yields S_2 in an incremental way.

Similar to the construction of slice-creating edit scripts, the basic idea is to consider the model M as model-creating edit script $\Delta_{\epsilon \Rightarrow M}$. The slice-updating edit script must delete all elements in the set $S_1 \setminus S_2$ from the current slice S_1, while adding all model elements in $S_2 \setminus S_1$. It is constructed as follows: Let P_{S_1} and P_{S_2} be the sets of rule applications which create all the elements in S_1 and S_2, respectively. Next, the sets P_{rem} and P_{add} of rule applications in $\Delta_{\epsilon \Rightarrow M}$ are determined with $P_{rem} = P_{S_1} \setminus P_{S_2}$ and $P_{add} = P_{S_2} \setminus P_{S_1}$. Finally, the resulting edit script $\Delta_{S_1 \Rightarrow S_2}$ contains (1) the rule applications in set P_{add}, with the same dependencies as in $\Delta_{\epsilon \Rightarrow M}$, and (2) for each rule application in P_{rem}, its inverse rule application with reversed dependencies as in $\Delta_{\epsilon \Rightarrow M}$. By construction, there cannot be dependencies between rule applications in both sets, so they can be executed in arbitrary order.

In addition to the completeness of the set R of transformation rules for a given modeling language (s. Sect. 4.2), *a second applicability condition is that, for each rule r in R, there must be an inverse rule r^{-1} which reverts the effect of r.* Given that these conditions are satisfied and a slice-updating edit script $\Delta_{S_1 \Rightarrow S_2}$ can be created, its application to S_1 indeed behaves according to the incremental slice update as in Definition 6. This is so because, by construction, none of the model elements in the intersection of S_1 and S_2 in M is deleted by the edit script $\Delta_{S_1 \Rightarrow S_2}$. Consequently, none of the elements in the intersection of C_1 and C_2 in M, which is a subset of $S_1 \cap S_2$, is deleted.

4.4 Implementation

The framework instantiation has been implemented using a set of standard MDE technologies on top of the widely used Eclipse Modeling Framework (EMF), which employs an object-oriented implementation of graph-based models in which nodes and edges are represented as objects and references, respectively. Edit scripts are calculated using the model differencing framework SiLift [21], which uses EMF Compare [22] in order to determine the corresponding elements in a pair of models being compared with each other. A matching determined by EMF Compare fulfills the requirements presented in Sect. 4.1 since EMF Compare (a) delivers 1:1-correspondences between elements, thus yielding an injective mapping, and (b) implicitly matches edges if their respective source and target nodes are matched and if they have the same type (because EMF does not support parallel edges of the same type in general), thus yielding an edge-preserving mapping. Finally, transformation rules are implemented using the model transformation language and framework Henshin [23,24] which is based on graph transformation concepts.

5 Solving the Motivating Examples

In this section, we outline the configurations of two concrete model slicers which are based on the framework instantiation presented in Sect. 4, and which are capable of solving the motivating examples introduced in Sect. 2. Each of these slicers is configured by a set of Henshin transformation rules which are used for the calculation of model-creating, and thus for the construction of slice-creating and slice-updating, edit scripts. The complete rule sets can be found at the accompanying website of this paper [25].

5.1 A State-Based Model Slicer

Two of the creation rules which are used to configure a state-based model slicer as described in our first example of Sect. 2 are shown in Fig. 7. The rules are depicted in an integrated form: the left- and right-hand sides of a rule are merged into a unified model graph following the visual syntax of the Henshin model transformation language [23].

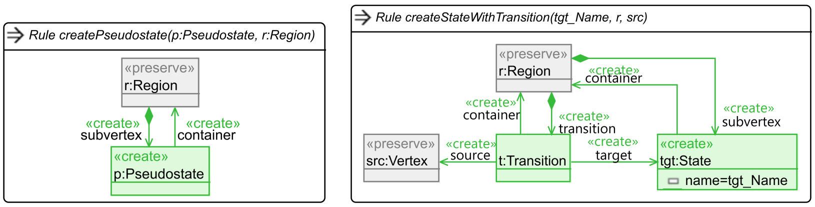

Fig. 7. Subset of the creation rules for configuring a state-based model slicer

Most of the creation rules are of a similar form as the creation rule *createPseudostate*, which simply creates a pseudostate and connects it with an existing container. The key idea of this slicer configuration, however, is the special creation rule *createStateWithTransition*, which creates a state together with an incoming transition in a

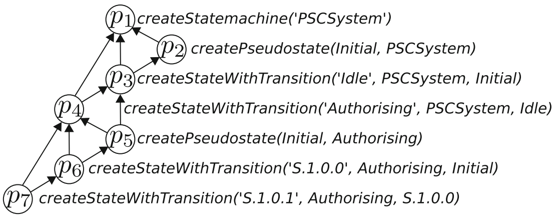
p_1 createStatemachine('PSCSystem')
p_2 createPseudostate(Initial, PSCSystem)
p_3 createStateWithTransition('Idle', PSCSystem, Initial)
p_4 createStateWithTransition('Authorising', PSCSystem, Idle)
p_5 createPseudostate(Initial, Authorising)
p_6 createStateWithTransition('S.1.0.0', Authorising, Initial)
p_7 createStateWithTransition('S.1.0.1', Authorising, S.1.0.0)

Fig. 8. Slice-creating edit script.

single step. To support the incremental updating of slices, for each creation rule an inverse deletion rule is included in the overall set of transformation rules. Parts of the resulting model-creating edit script using these rules are shown in Fig. 8. For example, rule application *p3* creates the state *Idle* in the top-level region of the state machine *PSCSystem*, together with an incoming transition having the initial state of the state machine, created by rule application *p2*, as source state. Thus, *p3* depends on *p2* since the initial state must be created first. Similar dependency relationships arise for the creation of other states which are created together with an incoming transition.

The effect of this configuration on the behavior of the model slicer is as follows (illustrated here for the creation of a new slice): If state *S.1.0.1* is selected as slicing criterion, as in our motivating example, rule application *p7* is included in the slice-creating edit script since it creates that state. Implicitly, all rule applications on which *p7* transitively depends on, i.e., all rule applications *p1* to *p6*, are also included in the slice-creating edit script. Consequently, the slice resulting from applying the slice-creating edit script to an empty model creates a submodel of the state machine of Fig. 1 which contains a transition path from its initial state to state *S.1.0.1*, according to the desired behavior of the slicer.

A current limitation of our solution is that, for each state *s* of the slicing criterion, only a single transition path from the initial state to state *s* is sliced. This path is determined non-deterministically from the set of all possible paths from the initial state to state *s*. To overcome this limitation, rule schemes comprising a kernel rule and a set of multi-rules (see, e.g., [26,27]) would have to be supported by our approach. Then, a rule scheme for creating a state with an arbitrary number of incoming transitions could be included in the configuration of our slicer, which in turn leads to the desired effect during model slicing. We leave such a support for rule schemes for future work.

5.2 A Slicer for Extracting Editable Submodels

In general, editable models adhere to a basic form of consistency which we assume to be defined by the effective meta-model of a given model editor [28]. The basic idea of configuring a model slicer for extracting editable submodels, adopted from [8], is that all creation and deletion rules preserve this level of consistency. Given an effective meta-model, such a rule set can be generated using the approach presented in [28] and its EMF-/UML-based implementation [29,30].

In our motivating example of Sect. 2, for instance, a consistency-preserving creation rule *createTrigger* creates an element of type *Trigger* and immediately connects it to an already existing operation of a class. The operation serves as the *callEvent* of this trigger and needs to be created first, which leads to a dependency in a model-creating edit script. Thus, if a trigger is included in the slicing criterion, the operation serving as *callEvent* of that trigger will be implicitly included in the resulting slice since it is created by the slice-creating edit script.

6 Related Work

A large number of model slicers has been developed. Most of them work only with one specific type of models, notably state machines [4] and other types of behavioral models such as MATLAB/Simulink block diagrams [5]. Other supported model types include UML class diagrams [31], architectural models [32] or system models defined using the SysML modeling language [33]. None of these approaches can be transferred to other (domain-specific) modeling languages, and they do not abstract from concrete slicing specifications.

The only well-known more generally usable technique which is adaptable to a given modeling language and slicing specification is Kompren [7]. In contrast to our formal framework, however, Kompren does not abstract from the concrete model modification approach and implementation technologies. It offers a domain-specific language based on the Kermeta model transformation language [34] to specify the behavior of a model slicer, and a generator which generates a fully functioning model slicer from such a specification. When Kompren is used in the so-called active mode, slices are incrementally updated when the input model changes, according to the principle of incremental model transformation [35]. In our approach, slices are incrementally updated when the slicing criterion is modified. As long as endogenous model transformations for constructing slices are used only, Kompren could be easily extended to become an instantiation of our formal framework.

Incremental slicing has also been addressed in [36], however, using a notion of incrementality which fundamentally differs from ours. The technique has been developed in the context of testing model-based delta-oriented software product lines [37]. Rather than incrementally updating an existing slice, the approach incrementally processes the product space of a product line, where each "product" is specified by a state machine model. As in software regression testing, the goal is to obtain retest information by utilizing differences between state machine slices obtained from different products.

In a broader sense, related work can be found in the area of model splitting and model decomposition. The technique presented in [38] aims at splitting a model into submodels according to linguistic heuristics and using information retrieval techniques. The model decomposition approach presented in [39] considers models as graphs and first determines strongly connected graph components from which the space of possible decompositions is derived in a second step. Both approaches are different from ours in that they produce a partitioning of an input model instead of a single slice. None of them supports the incremental updating of a model partitioning.

7 Conclusion

We presented a formal framework for defining model slicers that support incremental slice updates based on a general concept of model modifications. Incremental slice updates were shown to be equivalent to non-incremental ones. Furthermore, we presented a framework instantiation based on the concept of edit scripts defining application sequences of model transformation rules. This instantiation was implemented by two concrete model slicers based on the Eclipse Modeling Framework and the model differencing framework SiLift.

As future work, we plan to investigate incremental updates of both the underlying model and the slicing criterion. It is also worthwhile to examine the extent to which further concrete model slicers fit into our formal framework of incremental model slicing. For our own instantiation of this framework, we plan to cover further model transformation features such as rule schemes and application conditions, which will make the configuration of concrete model slicers more flexible and enable us to support further use cases and purposes.

Acknowledgments. This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future - Managed Software Evolution.

References

1. Weiser, M.: Program slicing. In: Proceedings of ICSE 1981. IEEE Press (1981)
2. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. ACM SIGSOFT Softw. Eng. Notes **30**(2), 1–36 (2005)
3. Brambilla, M., Cabot, J., Wimmer, M.: Model-driven software engineering in practice. Synth. Lect. Softw. Eng. **1**(1), 1–182 (2012)
4. Androutsopoulos, K., Clark, D., Harman, M., Krinke, J., Tratt, L.: State-based model slicing: A survey. ACM Comput. Surv. **45**(4), 36 (2013). https://doi.org/10.1145/2501654.2501667. Article 53
5. Gerlitz, T., Kowalewski, S.: Flow sensitive slicing for matlab/simulink models. In: Proceedings of WICSA 2016. IEEE (2016)
6. Samuel, P., Mall, R.: A novel test case design technique using dynamic slicing of UML sequence diagrams. e-Informatica **2**(1), 71–92 (2008)
7. Blouin, A., Combemale, B., Baudry, B., Beaudoux, O.: Kompren: modeling and generating model slicers. SoSyM **14**(1), 321–337 (2015)
8. Pietsch, C., Ohrndorf, M., Kelter, U., Kehrer, T.: Incrementally slicing editable submodels. In: Proceedings of ASE 2017. IEEE Press (2017)
9. Baker, P., Loh, S., Weil, F.: Model-driven engineering in a large industrial context—Motorola case study. In: Briand, L., Williams, C. (eds.) MODELS 2005. LNCS, vol. 3713, pp. 476–491. Springer, Heidelberg (2005). https://doi.org/10.1007/11557432_36
10. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: Proceedings of ICSE 2011. IEEE (2011)
11. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The grand challenge of scalability for model driven engineering. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 48–53. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01648-6_5
12. Kolovos, D.S., Rose, L.M., Matragkas, N., Paige, R.F., Guerra, E., Cuadrado, J.S., De Lara, J., Ráth, I., Varró, D., Tisi, M., et al.: A research roadmap towards achieving scalability in model driven engineering. In: Proceedings of BigMDE @ STAF 2013. ACM (2013)
13. Capozucca, A., Cheng, B., Guelfi, N., Istoan, P.: OO-SPL modelling of the focused case study. In: Proceedings of CMA @ MoDELS 2011 (2011)

14. Taentzer, G., Ermel, C., Langer, P., Wimmer, M.: Conflict detection for model versioning based on graph modifications. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 171–186. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15928-2_12
15. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
16. Habel, A., Pennemann, K.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. Comput. Sci. **19**(2), 245–296 (2009)
17. Kehrer, T., Kelter, U., Taentzer, G.: Consistency-preserving edit scripts in model versioning. In: Proceedings of ASE 2013. IEEE (2013)
18. Kolovos, D.S., Di Ruscio, D., Pierantonio, A., Paige, R.F.: Different models for model matching: an analysis of approaches to support model differencing. In: Proceedings of CVSM @ ICSE 2009. IEEE (2009)
19. Kehrer, T., Kelter, U., Pietsch, P., Schmidt, M.: Adaptability of model comparison tools. In: Proceedings of ASE 2011. ACM (2012)
20. Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: Proceedings of ASE 2011. IEEE (2011)
21. Kehrer, T., Kelter, U., Ohrndorf, M., Sollbach, T.: Understanding model evolution through semantically lifting model differences with SiLift. In: Proceedings of ICSM 2012. IEEE Computer Society (2012)
22. Brun, C., Pierantonio, A.: Model differences in the eclipse modeling framework. UPGRADE Eur. J. Inform. Prof. **9**(2), 29–34 (2008)
23. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_9
24. Strüber, D., Born, K., Gill, K.D., Groner, R., Kehrer, T., Ohrndorf, M., Tichy, M.: Henshin: a usability-focused framework for EMF model transformation development. In: de Lara, J., Plump, D. (eds.) ICGT 2017. LNCS, vol. 10373, pp. 196–208. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61470-0_12
25. Taentzer, G., Kehrer, T., Pietsch, C., Kelter, U.: Accompanying website for this paper (2017). http://pi.informatik.uni-siegen.de/projects/SiLift/fase2018/
26. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. I. World Scientific Publishing Co., Inc., River Edge (1997)
27. Biermann, E., Ermel, C., Taentzer, G.: Lifting parallel graph transformation concepts to model transformation based on the eclipse modeling framework. Electron. Commun. EASST **26** (2010)
28. Kehrer, T., Taentzer, G., Rindt, M., Kelter, U.: Automatically deriving the specification of model editing operations from meta-models. In: Van Van Gorp, P., Engels, G. (eds.) ICMT 2016. LNCS, vol. 9765, pp. 173–188. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42064-6_12
29. Rindt, M., Kehrer, T., Kelter, U.: Automatic generation of consistency-preserving edit operations for MDE tools. In: Proceedings of Demos @ MoDELS 2014. CEUR Workshop Proceedings, vol. 1255 (2014)
30. Kehrer, T., Rindt, M., Pietsch, P., Kelter, U.: Generating edit operations for profiled UML models. In: Proceedings ME @ MoDELS 2013. CEUR Workshop Proceedings, vol. 1090 (2013)

31. Kagdi, H., Maletic, J.I., Sutton, A.: Context-free slicing of UML class models. In: Proceedings of ICSM 2005. IEEE (2005)
32. Lallchandani, J.T., Mall, R.: A dynamic slicing technique for UML architectural models. IEEE Trans. Softw. Eng. **37**(6), 737–771 (2011)
33. Nejati, S., Sabetzadeh, M., Falessi, D., Briand, L., Coq, T.: A SysML-based approach to traceability management and design slicing in support of safety certification: framework, tool support, and case studies. Inf. Softw. Technol. **54**(6), 569–590 (2012)
34. Jézéquel, J.-M., Barais, O., Fleurey, F.: Model driven language engineering with Kermeta. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 201–221. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18023-1_5
35. Etzlstorfer, J., Kusel, A., Kapsammer, E., Langer, P., Retschitzegger, W., Schoenboeck, J., Schwinger, W., Wimmer, M.: A survey on incremental model transformation approaches. In: Pierantonio, A., Schätz, B. (eds.) Proceedings of the Workshop on Models and Evolution. CEUR Workshop Proceedings, vol. 1090, pp. 4–13 (2013)
36. Lity, S., Morbach, T., Thüm, T., Schaefer, I.: Applying incremental model slicing to product-line regression testing. In: Kapitsaki, G.M., Santana de Almeida, E. (eds.) ICSR 2016. LNCS, vol. 9679, pp. 3–19. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-35122-3_1
37. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15579-6_6
38. Struber, D., Rubin, J., Taentzer, G., Chechik, M.: Splitting models using information retrieval and model crawling techniques. In: Gnesi, S., Rensink, A. (eds.) FASE 2014. LNCS, vol. 8411, pp. 47–62. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54804-8_4
39. Ma, Q., Kelsen, P., Glodt, C.: A generic model decomposition technique and its application to the eclipse modeling framework. SoSyM **14**(2), 921–952 (2015)