Incrementally Slicing Editable Submodels

Christopher Pietsch, Manuel Ohrndorf, Udo Kelter
Software Engineering Group
University of Siegen
Germany
{cpietsch, mohrndorf, kelter}@informatik.uni-siegen.de

Timo Kehrer
Department of Computer Science
Humboldt-University of Berlin
Germany
{timo.kehrer}@informatik.hu-berlin.de

***Abstract*—Model slicers are tools which provide two services: (a) finding parts of interest in a model and (b) displaying these parts somehow or extract these parts as a new, autonomous model, which is referred to as slice or sub-model. This paper focuses on the creation of editable slices, which can be processed by model editors, analysis tools, model management tools etc. Slices are useful if, e.g., only a part of a large model shall be analyzed, compared or processed by time-consuming algorithms, or if sub-models shall be modified independently. We present a new generic incremental slicer which can slice models of arbitrary type and which creates slices which are consistent in the sense that they are editable by standard editors. It is built on top of a model differencing framework and does not require additional configuration data beyond those available in the differencing framework. The slicer can incrementally extend or reduce an existing slice if model elements shall be added or removed, even if the slice has been edited meanwhile. We demonstrate the usefulness of our slicer in several scenarios using a large UML model. A screencast of the demonstrated scenarios is provided at http://pi.informatik.uni-siegen.de/projects/SiLift/ase2017.**

I. Introduction

Document slicers are tools which provide two services: (a) finding "interesting" parts of an existing document and (b) displaying these parts in appropriate form or producing a new document which consists of these parts, which is referred to as slice or sub-document. Slicing of programs has a long tradition [1], [2]. Program slicers address a broad range of use cases where a program slice with certain properties is required. They put most emphasis on service (a): for a given, user-specified *slicing criterion*, they find related program fragments. These slicers can be regarded as highly specialized query processors. Service (b) is of minor importance here.

With the advent of model-based software development (MBSD), models rather than source code play the role of primary software artifacts. Many similar use cases as known from program slicing must be supported for model slicing. Model slicing is, additionally, often motivated by a different class of use cases where step (b), *the construction of a new document, is a main requirement and a big challenge*. One typical motivation are inefficient model checkers, test suite generators etc.; in order to reduce run-times, they are applied to small slices (or sub-models) rather than to a large complete model. Further motivations include transport of the slice and integration into other system models, or independent editing of parts of a model. These and similar motivations of model slicing are addressed by the slicer presented in this paper.

From a more technical point of view, producing a slice as a new model can be regarded as a function call

```
Slice aSlice = slice(largeModel, slCrit);
```

`slCrit` is the slicing criterion provided by the user. Internally, the computation of the slice roughly proceeds as follows:

```
ElementSet RME = select(largeModel, slCrit);
ElementSet extRME = extend(largeModel, RME);
Slice aSlice = createSl(largeModel, extRME);
```

The first task, *the selection*, determines the set `RME` of requested model elements (more specifically the sets of nodes and edges of the abstract syntax graph of `largeModel`) which must be present in the slice, e.g.,
– all direct and indirect super-classes of a given class "X",
– the sequence-diagram named "X" of a system model, or
– the set of model elements violating a given OCL inivariant.

In many cases, finding these requested model elements can be expressed as a simple graph query using languages such as OCL, XPath etc. The set `RME` does not normally conform to the required properties of the slice, e.g., conformance to the meta-model.

The second task, *the extension*, extends `RME` by further model elements to make it displayable and editable in standard visual editors for the given modeling language. For example, assume that `RME` comprises the elements of a sequence diagram of a large UML model and that label names of lifelines or messages are derived from class definitions. Then `extRME` must contain appropriate fragments of these class definitions.

A large number of model slicers has been developed. Most of them work only with one specific type of models, notably state machines and similar types of "executable" models [3], [4]. Other supported model types include class diagrams, feature models and meta-models. Virtually all proposed slicers only support the selection task. The selection function of these slicers typically exploits the semantics of the models. This implies that these approaches cannot be transferred to model types which have different semantics or no execution semantics at all. Moreover, they can only slice models with a certain degree of syntactic and semantic correctness, i.e., models which conform to all basic and many advanced (but not always all) constraints defined in their meta model.

The only well-known more generally usable technique which also supports the second step of generating a new model is Kompren [5], which is actually a generator for model slicers. A slicer generated by Kompren transforms a large model into a smaller one, the slice. Kompren does not provide task-specific queries for finding “interesting” parts of a model based on a “slicing criterion,” but provides only general querying facilities for models. The user must implement the search for interesting parts and specify how to extend these parts to a valid model by using a dedicated domain-specific language. The user is responsible to guarantee that a slice exhibits all desired properties. Moreover, slices are always created from scratch, incrementally adapting a slice, an indispensable feature for all use cases where slices are edited by developers, is not supported.

In this paper, we present a new generic slicer with the following novel features:

a) It is integrated within a suite of other modeling tools which all assume a common notion of “basic consistency” of models, which is basically given by the standard model editor. Technically, we refer to this consistency level as “effective meta model” [6] of the tool suite.

b) Our slicer guarantees that all slices produced conform to the effective meta model, i.e. they are editable with the standard model editor.

c) Our slicer does not need additional configuration data. It automatically configures itself using configuration data available in our tool suite.

d) Technically, our slicer is divided in two components: (a) a *server*, which accesses the whole model and which can run on a powerful server hardware, (b) a *client*, which runs on a workstation, maintains the slice and does not load or directly access the whole model.

e) Our slicer is incremental in the sense that a slice can be extended or reduced, more specifically, that arbitrary sets of model elements can be added to, or removed from, RME and the slice is automatically adapted. This adaptation is fully automated if the slice is unchanged. If the slice has been edited, possible conflicts are resolved interactively.

II. Problem Motivation

In this section, we introduce a running example and motivate the problem of slicing editable sub-models. Finally, we give a formal definition of the notion of incremental slicing.

A. Running Example

We will use the *Barbados Car Crash Crisis Management System (bCMS)* [7] as our running example. Figure 1 shows an excerpt of the whole system model describing the operations of a police and a fire department in case of a crisis situation. The system is modeled from different viewpoints. The class diagram models the static view of the system, i.e., its key entities and their relationships. State machine and sequence diagrams model the dynamic view of the system, i.e., its runtime behavior.

The different diagrams are not, as one might think, independent, in fact they depend on each other. Figure 2 shows an excerpt of the UML meta-model illustrating the interrelations

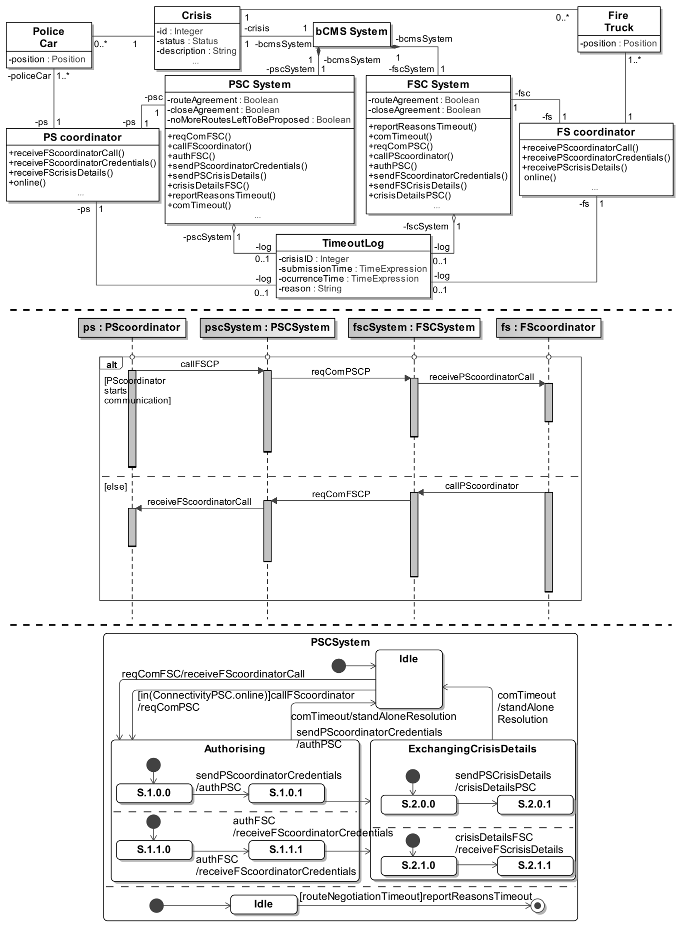

Police Car
-position : Position
Crisis
-id : Integer
-status : Status
-description : String
bCMS System
Fire Truck
-position : Position
PSC System
-routeAgreement : Boolean
-closeAgreement : Boolean
-noMoreRoutesLeftToBeProposed : Boolean
+reqComFSC()
+callFScoordinator()
+authFSC()
+sendPScoordinatorCredentials()
+sendPSCrisisDetails()
+crisisDetailsFSC()
+reportReasonsTimeout()
+comTimeout()
FSC System
-routeAgreement : Boolean
-closeAgreement : Boolean
+reportReasonsTimeout()
+comTimeout()
+reqComPSC()
+callPScoordinator()
+authPSC()
+sendFScoordinatorCredentials()
+sendFSCrisisDetails()
+crisisDetailsPSC()
PS coordinator
+receiveFScoordinatorCall()
+receiveFScoordinatorCredentials()
+receiveFScrisisDetails()
+online()
FS coordinator
+receivePScoordinatorCall()
+receivePScoordinatorCredentials()
+receivePScrisisDetails()
online()
TimeoutLog
-crisisID : Integer
-submissionTime : TimeExpression
-ocurrenceTime : TimeExpression
-reason : String
ps : PScoordinator
pscSystem : PSCSystem
fscSystem : FSCSystem
fs : FScoordinator
alt
[PScoordinator starts communication]
callFSCP
reqComPSCP
receivePScoordinatorCall
[else]
receiveFScoordinatorCall
reqComFSCP
callPScoordinator
PSCSystem
Idle
reqComFSC/receiveFScoordinatorCall
[in(ConnectivityPSC.online)]callFScoordinator /reqComPSC
comTimeout/standAloneResolution
comTimeout /standAlone Resolution
sendPScoordinatorCredentials /authPSC
Authorising
ExchangingCrisisDetails
S.1.0.0
S.1.0.1
S.1.1.0
S.1.1.1
authFSC /receiveFScoordinatorCredentials
S.2.0.0
S.2.0.1
sendPSCrisisDetails /crisisDetailsPSC
S.2.1.0
S.2.1.1
crisisDetailsFSC /receiveFScrisisDetails
[routeNegotiationTimeout]reportReasonsTimeout

Fig. 1. Excerpt of the system model of the bCMS case study [7].

of the different diagrams based on the abstract syntax. For instance, a `Lifeline` in a sequence diagram usually represents a `Property` of a `Class`. An `Operation` of a class can be used as the `signature` of a `Message` (s. green-shaded part in Figure 2). Analogously, a `CallEvent` that is used by a `Trigger` of a `Transition` in a state machine diagram must refer to an `Operation` of a `Class` (s. red-shaded part in Figure 2).

B. Extending the Set RME

Assume a developer wants to extract a state machine (or a part of it) from the complete model. In the abstract syntax graph (ASG) of our model, a state machine is basically a subtree of the containment tree which contains all states, transitions etc. of this state machine. RME would thus be defined as the set of all ASG nodes in this subtree. If we simply copy this subtree into a new empty model then the resulting ASG contains all nodes specified in RME, but violates the multiplicity constraint of the reference `event` between `Trigger` and `CallEvent` because the `CallEvent` is not part of the state machine (it is owned by the package), but a “mandatory neighbor”. `CallEvents` are typical examples of ASG nodes which implement a relationship between two sub-models. They have no corresponding graphical object in the visual representation of these sub-models and do not represent

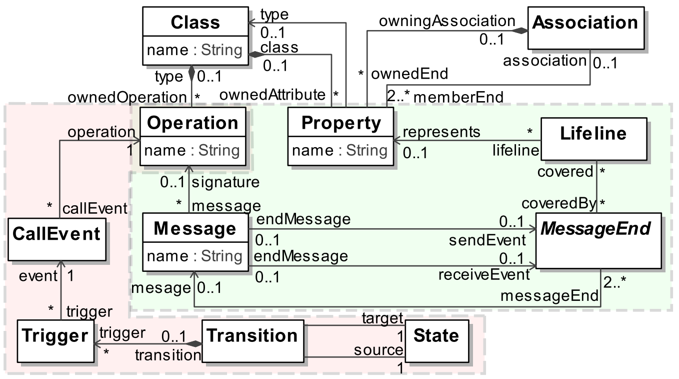


Fig. 2. Excerpt of the UML meta-model.

a user-level concept of the modeling language. An average user is hardly able or willing to manually deal with these ASG nodes. Without them, however, an ASG can be considered corrupted and will not be processed by an editor or analysis tool (s. Section V for further scenarios).

The main aim of the extension of RME *to a superset* extRME *is to add these technically required ASG nodes.*

C. Incremental Slicing

A slice initially created for a set of requested model elements $\texttt{RME}_{old}$ can turn out to be inappropriate, most typically because additional model elements are desired or because the slice is still too large. One option to get a better slice is to define a revised set RME, namely $\texttt{RME}_{new}$, to discard the old slice, and to recompute the slice for $\texttt{RME}_{new}$. This option does not work if the slice has been edited. A better approach is thus to obtain a model patch which, if applied to the current slice, adds or removes model elements as desired. If the slice is unchanged such a patch can be applied successfully without raising errors. If it has been edited, conflicts can occur, which must be resolved appropriately.

There are two basic approaches for defining the new slice. One approach is to use a new slicing criterion and to compute $\texttt{RME}_{new}$ as $\texttt{select(largeModel, slCrit}_{new}\texttt{).RME}_{old}$ and $\texttt{RME}_{new}$ implicitly define the sets of additional and removable model elements:

$\texttt{addRME} = \texttt{RME}_{new} \setminus \texttt{RME}_{old}$ and

$\texttt{remRME} = \texttt{RME}_{old} \setminus \texttt{RME}_{new}$.

Another approach is to let the user *explicitly* define the sets addRME and remRME by a suitable user interface. The resulting slice does not necessarily have a compact slicing criterion in the syntax accepted by the select function.

In both user interface designs, incrementally adapting an existing slice boils down to an (internal) operation call

aSlice.incrAdapt(addRME, remRME);

The effect of the incrAdapt operation is conceptually as follows. Our current slice actually contains the elements $\texttt{extend(RME}_{old}\texttt{)}$, the new slice should contain the elements $\texttt{extend(RME}_{new}\texttt{)}$. The model patch which adapts the slice to $\texttt{RME}_{new}$ must thus effectively remove the elements removedME and add the elements addedME, with

$\texttt{removedME} = \texttt{extend(RME}_{old}\texttt{)} \setminus \texttt{extend(RME}_{new}\texttt{)}$,

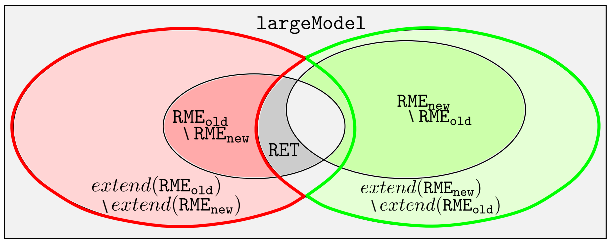

largeModel
$\texttt{RME}_{\texttt{old}} \setminus \texttt{RME}_{\texttt{new}}$
RET
$\texttt{RME}_{\texttt{new}} \setminus \texttt{RME}_{\texttt{old}}$
$extend(\texttt{RME}_{\texttt{old}}) \setminus extend(\texttt{RME}_{\texttt{new}})$
$extend(\texttt{RME}_{\texttt{new}}) \setminus extend(\texttt{RME}_{\texttt{old}})$

Fig. 3. Added and removed model elements

$\texttt{addedME} = \texttt{extend(RME}_{new}\texttt{)} \setminus \texttt{extend(RME}_{old}\texttt{)}$, s. red- and green-bordered areas in Figure 3. Note that elements in the set $\texttt{RET} = \texttt{RME}_{old} \cap \texttt{(extend(RME}_{new}\texttt{)} \setminus \texttt{RME}_{new}\texttt{)}$, although specified as *removable*, are retained, because they are required in $\texttt{extend(RME}_{new}\texttt{)}$ (s. dark gray part in Figure 3).

III. Basic Design Decisions

In this section, we present basic design decisions of our approach. We first introduce the distribution model of our tool architecture, before we present the basic idea of creating a new slice and adapting an existing slice based on the principle of document patching.

A. Distribution Model

If very large models are to be sliced for performance reasons it would be counter-productive if the complete model is required to be available in the workspace or even to be loaded.

One basic design decision of our approach is therefore to separate between a client and a server (s. Figure 4), which can be physically distributed over different machines. The complete large model resides on the server, typically in a centrally hosted model repository. The slice exists in a local workspace at the client site.

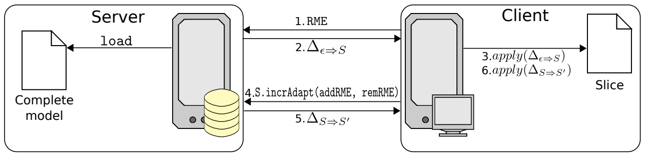

Server
load
Complete model
1.RME
$2.\Delta_{\epsilon \Rightarrow S}$
4.S.incrAdapt(addRME, remRME)
$5.\Delta_{S \Rightarrow S'}$
Client
$3.apply(\Delta_{\epsilon \Rightarrow S})$
$6.apply(\Delta_{S \Rightarrow S'})$
Slice

Fig. 4. Logical separation of our model slicer into client and server parts.

B. Creating a New Slice

A user requests the generation of a new slice S by specifying a set RME of requested model elements. Our approach is open to any method for determining this set. Our approach to construct S in the client is to (a) calculate a *creating patch* $\Delta_{\epsilon \Rightarrow S}$ on the server side, (b) send it to the client, and (c) apply it to the empty model ϵ. The resulting slice S contains all model elements in extRME (and thus the subset RME).

To that end, a basic design decision is to exploit existing technologies for calculating and handling of consistency-preserving edit scripts [8], a special form of model patches, as it will be explained in more detail later.

C. Adapting an Existing Slice

To adapt a slice `S`, the client sends an invocation of `S.incrAdapt(addRME, remRME)` to the server. The server (a) determines the sets of elements `removedME` and `addedME` as defined in Section II-C, (b) constructs an *adapting patch* $\Delta_{S \Rightarrow S'}$, and (c) sends $\Delta_{S \Rightarrow S'}$ to the client, where it is applied to the current slice `S`, which is transformed to `S'`. We utilize again consistency-preserving edit scripts here.

By construction of our edit scripts, updating an unmodified slice will never fail. However, if the slice has been edited in the client local changes can be in conflict with changes specified by the adapting patch. This problem also occurs when local copies of models in workspaces are to be updated by changes checked in into a repository. A solution to this problem, which is based on the principle of model patching, has been presented in [9]. This workspace update tool detects attempts to overwrite local changes and enables users to intervene and interactively handle the problem. It can be seamlessly integrated with our approach to incremental model slicing.

IV. Implementation

This section outlines the implementation of our approach, focusing on the server-side functionality for determining an extended model element set, and for generating slice-creating and slice-adapting edit scripts. We start by briefly recalling some basic background on consistency-preserving edit scripts.

A. Consistency-preserving Edit Scripts

Our approach uses consistency-preserving edit scripts [8] as a special form of model patches. An *edit script* $\Delta_{M1 \Rightarrow M2}$ obtained comparing two models $M1$ and $M2$ is a data structure which basically comprises a set of edit steps semiordered by an acyclic dependency graph in which the nodes are edit steps and the edges are dependencies between edit steps [8]. Applying $\Delta_{M1 \Rightarrow M2}$ to $M1$ yields $M2$. An *edit step* is an edit operation supplied with concrete arguments.

In our approach, *edit operations* are formally defined as in-place transformation rules on the ASG using the Henshin model transformation language and system [10], [11]. Edit operations must be consistency-preserving (CPEOs) in the sense that they transform a model from one consistent state to another. Consequently, an edit script based on CPEOs constructs a consistent model.

Each modeling language has its own set of CPEOs; we generate such a set from a meta-model using the (meta-)tool presented in [6], [12]. The meta-models we actually use reflect the actual consistency constraints enforced by the standard editor of an environment, they are referred to as *effective meta-model*. The effective meta-model differs from a "perfect" meta-model as defined in language standards such as the UML by removing many complex constraints, while all basic typing constraints and many multiplicities remain. Some multiplicity constraints may even be added [6]. For example, the effective meta-model of the UML modeling tool Papyrus enforces all multiplicity constraints shown in Figure 2. It tightens the multiplicities [0..1] for `sendEvent` and `receiveEvent` to [1..1] since a `Message` must always have a send and a receive end to be graphically displayable. Due to such multiplicity constraints, CPEOs typically create (or delete) *several* ASG nodes and related references as *one inseparable graph fragment*. For example, a message is created (deleted) together with its message ends.

A CPEO set R generated by our generator [6] is *complete* in the sense that every consistent model M of the DSML defined by the input meta-model can be constructed, starting from the empty model ϵ, by using creation CPEOs in R. A creation CPEO is specified by a rule which creates ASG elements only. Conversely, every consistent model M can be reduced to the empty model ϵ by using deletion CPEOs available in R.

For every creation CPEO r there is an inverse deletion CPEO r^{-1} such that, for every consistent model M of the given modeling language, applying the editing sequence $\langle r, r^{-1} \rangle$ yields M again.

B. Generating an Slice-Creating Edit Script

For the sake of didactics, the pseudo-code for creating a slice in Section I has suggested that, for a given set `RME`, `extRME` is computed first and then used to create the slice. Our approach actually reverses this order. The basic idea is to consider the large model L as created by an edit script $\Delta_{\epsilon \Rightarrow L}$ applied to an empty model ϵ and to identify a subset of $\Delta_{\epsilon \Rightarrow L}$ which creates all elements of `RME`. This subset is constructed as follows:

(1) An edit script $\Delta_{\epsilon \Rightarrow L}$ is created by comparing L with the empty model ϵ. This script contains only edit steps which create ASG fragments, i.e., each edit step refers to a creation CPEO. As long as the large model L remains unchanged, this edit script can be cached on the server to avoid its re-calculation for every client request.

(2) We identify the set ES_{RME} of edit steps in $\Delta_{\epsilon \Rightarrow L}$ which create at least one requested model element. Model elements appear as arguments in edit steps and can be easily found in $\Delta_{\epsilon \Rightarrow L}$. The ASG fragment created by an edit step can contain *further elements* not contained in `RME`.

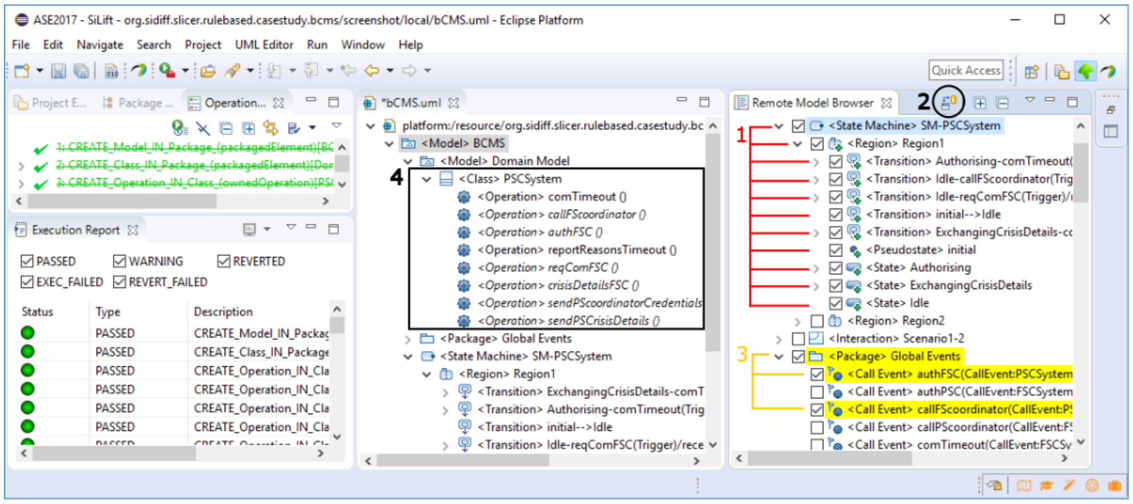
(3) We determine the set $ESpre_{RME}$ of all edit steps in $\Delta_{\epsilon \Rightarrow L}$ on which at least one edit step of ES_{RME} is directly or indirectly dependent, i.e., which must precede the edit steps in ES_{RME}.

Our edit script $\Delta_{\epsilon \Rightarrow S} \subseteq \Delta_{\epsilon \Rightarrow L}$ consists of the edit steps ES_{RME} and $ESpre_{RME}$ identified in steps (2) and (3). More formally, this edit script consists of the subgraph of the dependency graph of $\Delta_{\epsilon \Rightarrow L}$ containing all nodes of ES_{RME} and all required nodes and connecting edges according the transitive closure of the "required" relation, which is implied by dependencies between edit steps.

$\Delta_{\epsilon \Rightarrow S}$ creates, if applied to ϵ, a slice containing at least all elements of `RME`, but actually more elements, namely `extRME`. `extRME` is thus derived from $\Delta_{\epsilon \Rightarrow S}$, it is implicitly computed together with $\Delta_{\epsilon \Rightarrow S}$.

C. Generating a Slice-adapting Edit Script

If an existing slice S shall be adapted according to an operation invocation `S.incrAdapt(addRME, remRME)`, s.


ASE2017 - SiLift - org.sidiff.slicer.rulebased.casestudy.bcms/screenshot/local/bCMS.uml - Eclipse Platform
File Edit Navigate Search Project UML Editor Run Window Help
Quick Access
Project E... Package ... Operation...
Execution Report
*bCMS.uml
Remote Model Browser
1
2
3
4

~~1: CREATE_Model_IN_Package_(packagedElement)[BC~~
~~2: CREATE_Class_IN_Package_(packagedElement)[Dor~~
~~3: CREATE_Operation_IN_Class_(ownedOperation)[PS~~

☑ PASSED ☑ WARNING ☑ REVERTED
☑ EXEC_FAILED ☑ REVERT_FAILED

Status	Type	Description
●	PASSED	CREATE_Model_IN_Packag
●	PASSED	CREATE_Class_IN_Package
●	PASSED	CREATE_Operation_IN_Cla
●	PASSED	CREATE_Operation_IN_Cla
●	PASSED	CREATE_Operation_IN_Cla
●	PASSED	CREATE_Operation_IN_Cla

```
platform:/resource/org.sidiff.slicer.rulebased.casestudy.bc
  <Model> BCMS
    <Model> Domain Model
      <Class> PSCSystem
        <Operation> comTimeout ()
        <Operation> callFScoordinator ()
        <Operation> authFSC ()
        <Operation> reportReasonsTimeout ()
        <Operation> reqComFSC ()
        <Operation> crisisDetailsFSC ()
        <Operation> sendPScoordinatorCredentials
        <Operation> sendPSCrisisDetails ()
    <Package> Global Events
    <State Machine> SM-PSCSystem
      <Region> Region1
        <Transition> ExchangingCrisisDetails-comT
        <Transition> Authorising-comTimeout(Trig
        <Transition> initial-->Idle
        <Transition> Idle-reqComFSC(Trigger)/rece
```

```
☑ <State Machine> SM-PSCSystem
  ☑ <Region> Region1
    ☑ <Transition> Authorising-comTimeout(
    ☑ <Transition> Idle-callFScoordinator(Trig
    ☑ <Transition> Idle-reqComFSC(Trigger)/
    ☑ <Transition> initial-->Idle
    ☑ <Transition> ExchangingCrisisDetails-cc
    ☑ <Pseudostate> initial
    ☑ <State> Authorising
    ☑ <State> ExchangingCrisisDetails
    ☑ <State> Idle
  ☐ <Region> Region2
☐ <Interaction> Scenario1-2
☑ <Package> Global Events
  ☑ <Call Event> authFSC(CallEvent:PSCSystem
  ☐ <Call Event> authPSC(CallEvent:FSCSystem
  ☑ <Call Event> callFScoordinator(CallEvent:PS
  ☐ <Call Event> callPScoordinator(CallEvent:FS
  ☐ <Call Event> comTimeout(CallEvent:FSCSy
```

Fig. 5. Graphical user interface of the SiLift slicing environment.

Section II-C, the adapting edit script $\Delta_{S \Rightarrow S'}$ is constructed as follows. The sets ES_{RMEold} and $ESpre_{RMEold}$ of edit steps, which create `extend(RME`$_{old}$`)`, are assumed to be still available. For RME$_{new}$, the sets E_{RMEnew} and $ESpre_{RMEnew}$ are computed analogously. Let

– $ES_{extend(RME_{old})} = ES_{RMEold} \cup ESpre_{RMEold}$
– $ES_{extend(RME_{new})} = ES_{RMEnew} \cup ESpre_{RMEnew}$
– $ES_{rem} = ES_{extend(RME_{old})} \setminus ES_{extend(RME_{new})}$
– $ES_{add} = ES_{extend(RME_{new})} \setminus ES_{extend(RME_{old})}$

The edit script $\Delta_{S \Rightarrow S'}$ contains two sets of edit steps: (1) the edit steps in set ES_{add}, with the same dependencies as in $\Delta_{\epsilon \Rightarrow L}$, (2) for each edit step $es \in ES_{rem}$, its inverse edit step es^{-1}, with reversed dependencies as in $\Delta_{\epsilon \Rightarrow L}$. By definition, there cannot be dependencies between edit steps in both sets, so they can be executed in arbitrary order.

V. Case Study

In this section we demonstrate the usage of our slicer using the running example introduced in Section II-A. The implementation is based on the Eclipse Modeling technology stack and re-uses the differencing and patching facilities of the SiLift model versioning framework [8], [9], [13]. A download option and a screencast of the demonstrated scenarios are provided at the accompanying website [14].

A. Scenario 1: Retrieving a Fresh Slice from a Repository

The set `RME` can be specified in our tool with a tree-based model browser, which is shown on the right-hand side of Figure 5 and which operates on the server-side data. In the first scenario, we initially want the slice to contain the upper region of the state machine `PSCSystem` (s. Figure 1). We select the elements `<State Machine> SM-PSCSystem`, `<Region> Region1` and all of its children (s. Figure 5, red label "1"). Next, we create the slice by applying the synchronization button in the toolbar (s. Figure 5, black label "2"). The resulting edit script is opened in the interactive patch view of SiLift [9] (s. left-hand side of Figure 5) and can be applied onto the local UML model resource which is opened in the editor (s. center of Figure 5). Furthermore, the model browser selects and highlights all implicitly added model elements created by the edit script (s. Figure 5, yellow label "3"), i.e. all model elements in `extRME` $\setminus$ `RME`. In this scenario several `CallEvents` are added. Each event references an `Operation` of the class `<Class> PSCSystem`, thus the class and its respective operations are also added to the slice (not shown in the visible part of the model browser of Figure 5, but in the editor window in the center; black label "4").

B. Scenario 2: Extending an Unmodified Slice

To extend our initial slice, we can select further elements in the model browser and synchronize our local model again. A new edit script is calculated which contains only the edit steps creating the additional selected elements (and potentially further edit steps on which these edit steps depend). For instance, selecting the subtree rooted at the element `<Region> Region2` of the state machine `<State Machine> SM-PSCSystem` extends the statemachine such that we obtain the state machine shown in Figure 1.

C. Scenario 3: Reducing an Unmodified Slice

We can also reduce our slice by deselecting elements in the model browser. For instance, to revert our last modification, we can simply deselect the region `<Region> Region2` including all of its children. As explained in Section III-C, deselected elements will not be deleted if they are still implicitly selected. For instance, this is the case if we want to remove the `Operation authFSC()` of the `Class PSCSystem` from `RME`. Here, the `Trigger` of the `Transition` between the states `S.1.1.0` and `S.1.1.1` still refers to that operation via a `CallEvent`. Consequently, `authFSC()` can only be removed together with the `Trigger`. This can be achieved by explicitly deselecting the `Trigger`.

D. Scenario 4: Extending an Edited Slice

Scenario 4 assumes that the slice was edited as follows: We add a new `Operation reqComFSC(in severity:`

`Integer)` in the `Class PSCSystem`. After that, we change the reference `operation` of the `CallEvent` referred by the `Trigger` of the `Transition` between the states `Idle` and `Authorising` (cf. Figure 1). Subsequently, we remove the `Operation authFSC()` for the respective `Class`.

After this, we want to extend our modified slice by the sequence diagram shown in Figure 1. The application of the edit operation creating the message `reqComPSC` between the lifelines `fscSystem: FSCSystem` and `pscSystem: PSCSystem` would fail now due to the missing input parameter value `signature: Operation = "PSCSystem.reqComFSC()"`. Here, we can exploit the interactive workspace update facility of SiLift [9] by binding the new `Operation reqComFSC(in severity: Integer` as parameter value. After that, the edit operation can be successfully applied onto the modified local slice.

VI. Related Work

As already mentioned in the introduction, most existing work on model slicing [3], [4], [15]–[17] focuses on the selection step of the overall slicing process; the second step, i.e., producing a new model which contains the selected elements, has been only superficially addressed or neglected completely.

Exceptions of this are Kompren [5], which we already discussed in the introduction, and the work of Debreceni et al. [18], an approach for extracting view models from a larger source model using a query language based on EMF-IncQuery. Views are incrementally updated in response to changes in the source model. Editing of the view model is not supported. As with Kompren, the developer is responsible to guarantee that the result of a user-defined query is actually an editable sub-model.

The extraction of consistent sub-models from larger models has been addressed in the field of model decomposition, e.g., by the techniques presented in [19], [20]. However, they use a fixed notion of consistency and cannot be adapted to a modeling environment.

A model slicing approach which appears to work incrementally is presented in [21] in the context of testing model-based delta-oriented software product lines. However, their notion of incrementality is different from ours. Rather than incrementally adapting an extracted sub-model, the approach incrementally processes the product space of a product line. As in software regression testing, the goal is to obtain retest information by utilizing differences between state machine slices.

VII. Conclusion

In this paper, we presented a novel model slicer supporting the generation of autonomous sub-models from a given larger model. It automatically configures itself by exploiting existing configuration data of a modeling environment, and has the following distinguishing features; (i) it guarantees that extracted sub-models are editable in standard (visual) editors of that modeling environment, and (ii) it is capable of incrementally adapting edited slices. In our demo, we specified a set of model elements to be contained in the slice using an interactive repository browser. An interesting direction for future work is to integrate our slicer with a graph query processor or a dedicated processor for slicing criteria, which are complementary to our approach.

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] M. Weiser, “Program slicing,” in *Proc. of ICSE’81*. IEEE Press, 1981.
[2] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, “A brief survey of program slicing,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 2, 2005.
[3] K. Androutsopoulos, D. Clark, M. Harman, J. Krinke, and L. Tratt, “State-based model slicing: A survey,” *ACM Computing Surveys*, vol. 45, no. 4, 2013.
[4] T. Gerlitz and S. Kowalewski, “Flow sensitive slicing for matlab/simulink models,” in *Proc. of WICSA’16*. IEEE, 2016.
[5] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux, “Kompren: modeling and generating model slicers,” *Springer SoSyM*, vol. 14, no. 1, 2015.
[6] T. Kehrer, G. Taentzer, M. Rindt, and U. Kelter, “Automatically deriving the specification of model editing operations from meta-models,” in *Proc. of ICMT’16*. Springer, 2016.
[7] A. Capozucca, B. Cheng, N. Guelfi, and P. Istoan, “OO-SPL modelling of the focused case study,” in *CMA @ MoDELS’11*, 2011.
[8] T. Kehrer, U. Kelter, and G. Taentzer, “Consistency-preserving edit scripts in model versioning,” in *Proc. of ASE’13*. IEEE, 2013.
[9] T. Kehrer, U. Kelter, and D. Reuling, “Workspace updates of visual models,” in *Proc. of ASE’14*. ACM, 2014.
[10] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, “Henshin: advanced concepts and tools for in-place emf model transformations,” in *Proc. of MoDELS’10*. Springer, 2010.
[11] D. Strüber, K. Born, K. D. Gill, R. Groner, T. Kehrer, M. Ohrndorf, and M. Tichy, “Henshin: A usability-focused framework for emf model transformation development,” in *Proc. of ICGT’17*, 2017.
[12] M. Rindt, T. Kehrer, and U. Kelter, “Automatic generation of consistency-preserving edit operations for MDE tools,” in *Demos @ MoDELS’14*, ser. CEUR Workshop Proceedings, vol. 1255, 2014.
[13] T. Kehrer, U. Kelter, M. Ohrndorf, and T. Sollbach, “Understanding model evolution through semantically lifting model differences with silift,” in *Proc. of ICSM’12*. IEEE Computer Society, 2012.
[14] U. of Siegen, Software Engineering Group, “Incremental Slicer - Tool Demonstration and Update Site,” 2017. [Online]. Available: http://pi.informatik.uni-siegen.de/projects/SiLift/ase2017/index.php
[15] H. Kagdi, J. I. Maletic, and A. Sutton, “Context-free slicing of uml class models,” in *Proc. of ICSM’05*. IEEE, 2005.
[16] J. T. Lallchandani and R. Mall, “A dynamic slicing technique for uml architectural models,” *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, 2011.
[17] 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*, vol. 54, no. 6, pp. 569–590, 2012.
[18] C. Debreceni, Á. Horváth, Á. Hegedüs, Z. Ujhelyi, I. Ráth, and D. Varró, “Query-driven incremental synchronization of view models,” in *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. ACM, 2014, p. 31.
[19] Q. Ma, P. Kelsen, and C. Glodt, “A generic model decomposition technique and its application to the eclipse modeling framework,” *Springer SoSyM*, vol. 14, no. 2, 2015.
[20] B. Carré, G. Vanwormhoudt, and O. Caron, “From subsets of model elements to submodels,” *Springer SoSyM*, vol. 14, no. 2, pp. 861–887, 2015.
[21] S. Lity, T. Morbach, T. Thüm, and I. Schaefer, “Applying incremental model slicing to product-line regression testing,” in *Intl Conf. on Software Reuse*. Springer, 2016.