Heterogeneous Megamodel Slicing for Model Evolution

RICK SALAY, SAHAR KOKALY, MARSHA CHECHIK AND TOM MAIBAUM

ME 2016 @ MODELS’16
OCT. 2, 2016, SAINT MALO, FRANCE
Motivation

Model slicing to identify change impact is a key technique for supporting model evolution.

![Diagram showing model slicing and potential change impact]

- **Model**
  - change (slicing criterion)
  - potential change impact (slice)
Motivation

Slicing is well studied for individual models ..

.. but not for heterogeneous collections of related models (megamodels) which are common in large projects

(deep) megamodel slicing can be useful for identifying impact due to evolution across multiple models
A Model Management Approach for Assurance Case Reuse due to System Evolution

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

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

Valentin Cassano  
McMaster Centre for Software Certification  
McMaster University  
Hamilton, Canada  
cassanv@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

Evolution in software systems is a necessary activity that occurs due to fixing bugs, adding functionality or improving system quality. Systems often need to be shown to comply with regulatory standards. Along with demonstrating compliance, an artifact, called an assurance case, is often produced to show that the system indeed satisfies the property imposed by the standard (e.g., safety, privacy, security, etc.). Since each of the system, the standard, and the assurance case can be presented as a model, we propose the extension and use of traditional model management operators to aid in the reuse of parts of the assurance case when the system undergoes an evolution. Specifically, we present a model management approach that eventually produces a partial evolved assurance case and guidelines to help the assurance engineer in completing it. We demonstrate how our approach works on an automotive subsystem regulated by the ISO 26262 standard.

Keywords  
Evolution, reuse, model management, regulatory compliance, assurance cases, certification.

1. INTRODUCTION

The pervasiveness of software in all aspects of human activity has created special concerns regarding issues such as safety, security and privacy. Governments and standard organizations (e.g., ISO) have responded to this trend by creating regulations and standards that software must comply with. Compliance is a complex and costly goal to achieve. They may have to comply with multiple standards due to multiple jurisdictions and track the changes in standards. Assurance cases – collections of arguments and evidence to support the claims of compliance – must be developed and managed. Finally, maintaining families of related software products further multiplies the effort. Increasingly, models and model-driven engineering are being used as means to facilitate communication and collaboration between the stakeholders in the compliance value chain and further to introduce automation into regulatory compliance tasks.

In a position paper [21], we laid out a research agenda for applying model management to address the software compliance problem and sketched its use in particular compliance management scenarios. In this paper, we focus on one of these scenarios – assurance case reuse due to system evolution – and develop it in detail. Fig. 1 illustrates the scenario at a high level. Assume that a current specification $S$ describes the specification for the software in a vehicle. In addition, a type of assurance case $A$, called a safety case, has been developed complying with the ISO 26262 vehicle functional safety standard [16]. Safety case $A$ contains perhaps thousands of safety claims about different components of the vehicle, as well as arguments and evidence to support these claims. Now if $S$ is evolved to $S'$ – for example, as a result of a new requirement or a bug fix – a corresponding safety case $A'$ for $S'$ must be developed. Due to complexity and effort required to develop a safety case, there is strong incentive to reuse as much of $A$ as possible in the creation of $A'$. We address this problem using a model management strategy. Specifically, we make the following contributions:

1. We define a generic model management framework for supporting assurance case reuse.

---

Friday, Oct 7 in the “Testing and Analysis” Session
Assurance Case Impact Assessment algorithm [MODELS’16]

**Params:** \(<\text{Slice}_T \ ; \ \text{Merge}_T>\)

**Input:** initial spec \(S : T\), assurance case \(A : AC\), traceability map \(R\), changed spec \(S' : T\), delta \(D = <C0a;C0d;C0c>\)

**Output:** Impact set estimate \(A_{\text{RMM}}\), impact kind annotation \(k_{\text{RMM}}\)

1: \(R'_A \leftarrow \text{Restrict}(R, D)\)
2: \(dc \leftarrow \text{Slice}_T(S, \text{Merge}_T(d,c))\)
3: \(ac \leftarrow \text{Slice}_T(S', \text{Merge}_T(a,c))\)
4: \(C2_{\text{recheck}} \leftarrow \text{Merge}_{AC}(\text{Trace}(R, dc), \text{Trace}(R'_A, ac))\)
5: \(C2_{\text{revise}} \leftarrow \text{Trace}(R, d)\)
6: \(C3_{\text{revise}} \leftarrow \text{Slice}_{AC}(M, C2_{\text{revise}})\)
7: \(C3_{\text{recheck}} \leftarrow \text{Slice}_{AC}(M, C2_{\text{recheck}})\)
8: \(A_{\text{RMM}} \leftarrow \text{Merge}_{AC}(C3_{\text{revise}}, C3_{\text{recheck}})\)
9: \(k_{\text{RMM}}(C3_{\text{recheck}}) \leftarrow \text{‘recheck’}\)
10: \(k_{\text{RMM}}(C3_{\text{revise}}) \leftarrow \text{‘revise’}\)
11: return \(A_{\text{RMM}}, k_{\text{RMM}}\)
Proposed Slicing Algorithm

✓ Operates on Megamodels: general modeling technique to represent collections of interrelated models

✓ Works with arbitrary model types (e.g., conceptual, behavioural, goal models, test models, etc.)

✓ Uses widely adopted notion of traceability relations to assess change impact between models.

Assumptions:

1. (Slicers) There is a slicer available for each model type represented in the megamodel

2. (Dependencies) The relationships express all and only the inter-model dependencies
A slicer for each model type is assumed to be available.
Slicing algorithm

criterion megamodel fragment

megamodel

apply individual slicers

union slices

propagate slices

exit when no more change

slice megamodel fragment
Example run: slicing criterion

apply individual slicers
union slices
propagate slices
Example run: 1\textsuperscript{st} iteration

- apply individual slicers
- union slices
- propagate slices

Diagram:
- CD
- SM
- DD

Graphical representation of the process flow.
Example run: 1\textsuperscript{st} iteration

- apply individual slicers
- union slices
- propagate slices

Diagram:
- CD
- SM
- DD

11
Example run: 1st iteration

- Apply individual slicers
- Union slices
- Propagate slices
Example run: 2\textsuperscript{nd} iteration

- apply individual slicers
- union slices
- propagate slices

Diagram:
- CD
- SM
- DD
- CD
- SM
- CD
- SM

13
Example run: 2\textsuperscript{nd} iteration

apply individual slicers
union slices
propagate slices

CD
SM
DD
CD
SM
CD
SM
Example run: 2\textsuperscript{nd} iteration
Example run: 3rd iteration
Example run: 3rd iteration

apply individual slicers
union slices
propagate slices

No change
DONE!

<table>
<thead>
<tr>
<th>CD</th>
<th>SM</th>
<th>DD</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD</td>
<td>SM</td>
<td>DD</td>
</tr>
<tr>
<td>CD</td>
<td>SM</td>
<td>DD</td>
</tr>
</tbody>
</table>

17
Post processing

The algorithm focuses only on propagating dependencies
  ◦ ..and requires each slicer to do the same

However, the slicing algorithm should produce an output having additional characteristics:
  ◦ (Well-formedness) Should consist of well-formed slice models
  ◦ (Referential integrity) Should contain enough information to resolve references between related models

These are obtained by expanding the algorithm output in a post-processing step.
Example: Power Sliding Door

Automotive subsystem that controls the behaviour of a power sliding door in a car.

- This example is presented in Part 10 of the ISO 26262 standard
Power sliding door architecture (before evolution)

Evolution scenario: Redundant Switch is removed
Power door slider architecture (after evolution)
Initial megamodel

PowerSlidingDoor: CD

- requestDoorOpen
- requestDoorClose
- communicatedWith: Driver Switch
- communicatedWith: Actuator
- activated: Boolean
- requestSpeed
- sensed_speed

PowerSlidingDoor: SD

- requestDoorOpen
- requestDoorClose
- requestSpeed
- sensed_speed

Driver Switch
- requestDoorOpen
- requestDoorClose
- sensed_speed

Actuator
- requestSpeed
- sensed_speed

Redundant Switch
- requestSpeed
- sensed_speed

Power Sliding Door: CD

vs_ecu: VS ECU
- requestDoorOpen
- requestDoorClose

Power Sliding Door: SD

ac_ecu: AC ECU
- requestSpeed
- sensed_speed

a: Actuator
- requestSpeed
- powered
- activated

ds: Driver Switch
- requestDoorOpen
- requestDoorClose

s: Redundant Switch
- requestSpeed
- sensed_speed

Power Sliding Door: CD

- communicatedWith: Power Sliding Door: SD

Power Sliding Door: SD

- communicatedWith: Power Sliding Door: CD
- requestDoorOpen
- requestDoorClose
- requestSpeed
- sensed_speed
- powered
- activated

if ac_ecu.sensed_speed<=15 and a.powered and s.closed
a.openDoor()
## Slicing rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Component under assessment</th>
<th>Dependant parts potentially impacted</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD1</td>
<td>Class</td>
<td>Owned attributes and methods. Associations connected to class. Attributes/methods in other classes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>using types introduced in this class. Subclasses.</td>
</tr>
<tr>
<td>SD1</td>
<td>Term (portion of an expression)</td>
<td>Associated expression.</td>
</tr>
<tr>
<td>SD2</td>
<td>Expression (guard/action)</td>
<td>Associated message.</td>
</tr>
<tr>
<td>SD3</td>
<td>Message</td>
<td>Associated arrow (from source to target lifeline).</td>
</tr>
<tr>
<td>SD4</td>
<td>Arrow</td>
<td>Arrows directly after the arrow in the sequence. Message on the arrow.</td>
</tr>
<tr>
<td>SD5</td>
<td>Lifeline</td>
<td>Arrows connected to the lifeline. Messages on arrows connected to the lifeline.</td>
</tr>
</tbody>
</table>
1st iteration: Slicing Criterion
1\textsuperscript{st} iteration: Level 1 slice

PowerSlidingDoor: CD

- Driver Switch
  - requestDoorOpen()
  - requestDoorClose()
  - communicatesWith: VS ECU

- VS ECU
  - getSpeed(sensed_speed)
  - sensed_speed: Real
  - communicatesWith: AC ECU

- AC ECU
  - requestSpeed()
  - sensed_speed: Real
  - communicatesWith: Actuator

- Actuator
  - powered: Boolean
  - opensDoor() (or closeDoor())
  - communicatesWith: Door

- Door
  - open: Boolean
  - controls: Actuator

- Redundant Switch
  - request Speed()
  - closed: Boolean
  - sensed_speed: Real

PowerSlidingDoor: SD

- vs_ecu: VS ECU
- ac_ecu: AC ECU
- a: Actuator
- ds: Driver Switch
- s: Redundant Switch

- requestDoorOpen()
- requestDoorClose()

- getSpeed(sensed_speed)
  - sensed_speed: Real

- requestSpeed()
  - sensed_speed: Real

- openDoor()
  - powered: Boolean
  - activated: Boolean

- closeDoor()
  - powered: Boolean
  - activated: Boolean

- requestSpeed()
  - sensed_speed: Real

- if sensed_speed <= 15
  - powered: Boolean
  - activated: Boolean

- if sensed_speed <= 15 and powered: Boolean
  - activated: Boolean
1\textsuperscript{st} iteration: Level 2 slice
1st iteration: Result
2\textsuperscript{nd} iteration: Level 1 slice
2\textsuperscript{nd} iteration: Result
3rd iteration: No change - done
Post processing: well-formedness and referential integrity
Analysis of the algorithm

Correctness and Termination

- Given the assumptions, the algorithm is shown to be correct and guaranteed to terminate.

Worst Case Time complexity

- \( O(N_a \times N_M \times SL(N_a) + N_M^2 \times N_a^2) \)
- Where \( N_a \) = # of atoms, \( N_M \) = # of models, \( SL(n) \) is upper bound of slicer time complexity when input has \( n \) atoms.

Minimality

- Algorithm is shown to produce the minimal slice containing all atoms dependent on the slicing criterion.
Summary of Contributions

Proposed: a slicing algorithm for heterogeneous megamodels that:

- works with collections of arbitrary model types by using existing type-specific model slicers
- uses standard model relationships (e.g. traceability relationships) for inter-model dependencies

Proved: algorithm is minimal, correct, terminates

Showed: algorithm has quadratic time complexity relative to slicer time complexity

Demonstrated: algorithm on a simple case study from an existing standard (ISO 26262)
Current and Future Work

We plan to generalize the algorithm to address the following:

- N-ary relationships
  - currently on binary relationships are supported
- Nested megamodels
  - currently the input megamodel cannot contain other megamodels
- Arbitrary relationship types
  - currently only dependency relationships are allowed – others: e.g., refinement, overlap, etc.

We are developing an implementation using the Model Management INTeractive (MMINT)* framework.

*https://github.com/adisandro/MMINT/
Questions?

Heterogeneous Megamodel Slicing for Model Evolution
RICK SALAY, SAHAR KOKALY, MARSHA CHECHIK AND TOM MAIBAUM

ME 2016 @ MODELS’16
OCT. 2, 2016, SAINT MALO, FRANCE