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

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

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

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, known as an assurance case, is often used 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. For companies, 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 assurance case reuse due to model evolution.
2. We identify and specify the model management operators needed for a semi-automated solution to the assurance case reuse problem and present an algorithm for reuse.
3. We evaluate the generic framework and proposed solution by instantiating it for ISO 26262 vehicle safety cases with the KAOS goal modeling language [6] used for expressing assurance cases. We then apply this instantiation to an automotive subsystem, namely, a power sliding door system.
The rest of this paper is structured as follows: Section 2 presents some background and preliminaries needed for our approach and Section 3 introduces assurance cases and related concepts. Section 4 discusses our generic assurance case reuse framework, where we present an algorithm that is then evaluated in Section 5. Section 6 is an application of our generic framework on the power sliding door example. Section 7 discusses related work, and Section 8 ends with a summary and future work.

2. BACKGROUND AND PRELIMINARIES

2.1 Modeling and Model Management

In model-driven engineering (MDE), the MOF (MetaObject Facility) specification [26] states that a metamodel consists of element types. Each element type has zero or more reference and attribute types. Given a model $M$ of a metamodel $T$, an atom of $M$ denotes any element, reference or attribute in $M$ and atoms$_M$ denotes the set of all atoms in $M$. For example, in a UML class diagram, Class is an element, OwnedBy is a reference and IsAbstract is an attribute.

A complexity problem in MDE arises due to the proliferation of software models. As such, the area of Model Management [1] has emerged to address this challenge. Model management focuses on a high-level view in which entire models and their relationships (i.e., mappings between models) can be manipulated using operators (i.e., specialized model transformations) to achieve useful outcomes. Model management operators that have been studied include match [1], diff [1], lift [27], and the ones we will use in this paper; slice [24] and merge [4].

The slice operator accepts a model and a slicing criterion and extracts the subset of the model satisfying the criterion. Model slicing is a way to manage model complexity by focusing on a relevant subset of a model.

The merge operator accepts two models and a relationship expressing the overlap between them and produces a model that combines the content of the models according to the overlap. Model merge must address the issue of conflicts that could occur when the content is combined.

To help visualize and work with collections of models and their relationships, model management uses a special type of model called a megamodel [9] whose elements represent models and links between elements represent relationships between the models. Operators for megamodel management, namely, filter, map, and reduce, are presented in [28].

Each of these model management operators can be viewed as an abstract transformation that defines a class of concrete transformations, i.e., the implementations that refine the operator for particular model types. For example, a model merge of class diagrams is implemented differently than a model merge of state machines. Another widely used class of transformations used in model management is bidirectional transformations [11], aimed to keep two related models synchronized when one of the models changes (e.g., via model co-evolution, correction, etc.) by generating the update for the other model.

2.2 Model Evolution

In MDE, model evolution is studied in order to understand why models change and how that impacts consistency of related models. Examples of kinds of model evolution changes are presented in a survey on the evolution of UML models [20]. In this paper, we are interested in three of the presented types: evolution due to fixing errors, evolution due to changing functionality, and evolution due to changing model quality.

The general approach we use is to determine what parts of the assurance case are impacted by the change. Then the new assurance case must redo these parts and potentially can reuse the unimpacted parts. Depending on the type of change we are considering, the impact assessment is different. In the case of fixing errors, this means that the current assurance case was either incorrect or incomplete (or both) because it did not catch the error. This points to the need to address two questions: (1) why the assurance case was not adequate, and (2) how to change the assurance case to address this type of change. For the former, this requires an analysis of the process followed to produce the original assurance case and a decision on its causes. We consider this to be outside of the scope of this paper. For the latter, this requires an assessment of the impact of the change in the system and then the corresponding impact in the assurance case. The impact of the complete/incorrect parts of the assurance case also need to be determined if these are different than the impact of the error fix. In the case of changing functionality, this means that the requirements must have also changed. Thus we must do an impact analysis of both the changed requirements and of the system changes and how these correspondingly impact the assurance case. Finally, in the case of changing model quality, this means that the existing system was adequate, so the assurance case was not flawed and the requirements have not changed. In this case, we just assess the impact of the change in the system and the corresponding impact on the assurance case.

3. ASSURANCE CASES

Quality standards mandate the creation of quality-specific requirements and assurance cases. For example, ISO 26262 describes how safety requirements, levels of specification and a safety case for these must be produced to certify the safety of a vehicle. Fig. 2 shows a simplified view of software development when assurance is considered. Here, “P” represents the quality of interest to be assured, e.g., safety, privacy, security, etc. For the given quality, the system requirements are determined and traced to the implementation through a series of specification levels that can include both requirements and design refinements. An assurance case for such a process must contain a validation argument for the initial requirements as well as verification arguments for each refinement step.

3.1 Modeling Assurance Cases

An assurance case is an artifact that shows how important claims about the system (e.g., requirement satisfaction) can be argued for, ultimately from evidence obtained about the system such as test results, expert opinion, etc. Several approaches to modeling assurance cases have been proposed, including GSN [18], CAE [2], KAOS-based [3] and, more recently, SACM [7]. All of these approaches agree that an assurance case must contain three core concepts: claims, ar-
arguments and evidence. In order to develop a generic model management framework for assurance case reuse, we use the abstract metamodel shown in Fig. 3 for an assurance case based on these core concepts rather than choosing a particular concrete approach from the literature.

A **Claim** represents a statement about the system or some part of it. The **state** attribute represents the truth state (e.g., true, affirmed, refuted, etc.) of this statement. An **Evidence** element represents some set of data obtained about the system. These could include test results, analysis results, an expert opinion, a formal correctness proof, etc. Here, the **state** attribute indicates the validity state of the evidence - e.g., currently valid, is stale and must be regenerated, etc. The **Argument** elements connect claims to each other and to evidence. An argument takes zero or more claims and evidence as input and has one claim as a conclusion. Semantically, it represents how the conclusion follows from the input claims and evidence.

There is natural derived dependency relation connecting atoms of an assurance case.

**Definition 1 (Assurance Case Dependency Relation).** Given an assurance case \( A : AC \) defined according to the metamodel in Fig. 3, the dependency relation \( ACdep \subseteq \text{atoms}_A \times \text{atoms}_A \) for all atoms \( x, x', x'' \in A \) is defined as follows:

- (reflexive) \( ACdep(x, x) \);
- if \( x \) is a **Claim** that is the **conclusion** of **Argument** \( x' \) then \( ACdep(x, x') \);
- if \( x \) is an **Argument** that has a **premise** or **Evidence** \( x' \) then \( ACdep(x, x') \);
- (transitive) if \( ACdep(x, x') \) and \( ACdep(x', x'') \) then \( ACdep(x, x'') \).

Furthermore, we make the following semantic assumptions about assurance cases:

- If assurance case \( A : AC \) is considered to be complete and correct then the truth state of claim \( c \) can only be affected by the truth state of some claim \( x \) or by the content of the evidence \( x \) if \( ACdep(c, x) \).
- If the truth state of input claims and the content of evidence for an argument do not change then the truth state of the conclusion claim cannot change.

4. A GENERIC ASSURANCE FRAMEWORK FOR MODEL EVOLUTION

In this section, we develop a generic model management-based framework for assurance case reuse in the context of system evolution.

4.1 Objective of Reuse

The objective of the assurance case reuse problem due to the system evolution can be stated as follows:

(Objective) Given system specification \( S \) with complete and correct assurance case \( A \), if \( S \) evolves to \( S' \), determine the **maximal reuse** of \( A \) to produce a complete and correct assurance case \( A' \) for \( S' \).

Note that we take a “complete and correct assurance case” to mean one that is acceptably complete and correct by the organization developing the system. The goal of finding the maximal reuse can be refined into two subgoals:

1. identify the **impact set** \( A_{S \rightarrow S'} \) – the subset of \( A \) impacted by the change in \( S \); and
2. identify the kind of impact for the atoms within the impact set.

Goal 1 implies that atoms of \( A \) outside the impact set can be reused within \( A' \) since they are not impacted by the change; thus, the impact set implicitly defines the maximal subset of \( A \) that can be reused. The relevance of Goal 2 is that there are two possible types of impact to an atom due to a change, and this affects the degree of reuse:

1. The change may affect the truth state of a claim or the validity of a piece of evidence. Thus, the claim/evidence can be reused directly but its truth/validity state must be rechecked.
2. The change may affect the definition of a claim, argument or piece of evidence and hence affect its interpretability. Thus, the claim, argument or piece of evidence must first be revised and then additionally, in the case of claim/evidence, its truth/validity state must be checked.

A Type 1 impact requires less effort because the claim/evidence can be reused directly (i.e., no revision) and rechecking can
sometimes be automated. For example, rechecking a test-based evidence involves re-running the test cases, rechecking a claim containing a formally specified property may be rechecked using a property checker, etc. Thus, a Type 1 impact exhibits greater reuse than a Type 2 impact.

For example, assume that the assurance case \( A \) contains the claim that the following property holds for the AutoLight subsystem that controls a vehicle headlight:

\( (P1) \) If the ambient light sensor detects less than 25 lumens then the head lights turn on.

Furthermore, assume that this claim has been verified using a set of test results as evidence. Now if the subsystem is evolved so that it uses a new algorithm to turn the head lights on, then this claim is impacted because its truth state may be affected, and the property needs to be rechecked. Furthermore, the evidence used previously is no longer valid and the tests must be re-performed to check the property. Now consider another evolution of the subsystem in which the ambient light sensor is removed since some other approach to detecting light is used. In this case, the definition of the claim is affected and the property \( P1 \) can no longer be properly interpreted since there is no ambient light sensor. In this case the claim must be first revised based on the new design to be interpretable and then have its truth state checked. Similarly, a revision to the test cases producing the evidence is required.

Identifying the impact set and kinds of impacts represents the ideal behaviour for an assurance case impact assessment approach. We define an actual impact assessment approach as follows:

**Definition 2 (Impact assessment approach).** Given a system specification \( S \) with a complete and correct assurance case \( A \), if \( S \) evolves to \( S' \) then an impact assessment approach \( R \) can be applied as \( R(S, A, S') \) to produce a pair \( (A_R, k) \) where \( A_R \) is the impact set estimate and function \( k : A_R \rightarrow \{\text{revise, recheck}\} \) identifies the impact kind annotation for the atoms of \( A_R \).

We can evaluate an impact assessment approach by comparing it to the ideal case.

**Definition 3 (Soundness and relative efficiency).** Given a system specification \( S \) with a complete and correct assurance case \( A \), and \( S \) evolves to specification \( S' \), (Soundness) Impact assessment approach \( R \) is sound if \( A_{S\rightarrow S'} \subseteq A_R \).

(Relative efficiency) A sound impact assessment approach \( R \) is relatively more efficient than \( R' \) iff \( A_R \subseteq A'_{R'} \).

Soundness is a correctness criterion for an impact assessment approach – a sound approach guarantees that all actually impacted atoms of the assurance case are identified by the approach. Relative efficiency is a quality criterion, and greater efficiency means that it finds fewer "false positives" (i.e., when it says an atom is impacted although it is not) and hence facilitates greater reuse.

### 4.2 The Framework

We have defined the objective of reuse and how to evaluate a possible impact assessment approach. We now define a generic assurance case evolution framework RMM based on model management that addresses reuse. The framework is generic because, as is typical with model management, it is defined independently of the specific model types used in an evolution scenario. Thus, applying it to a particular evolution case requires instantiating it for the model types used.

Fig. 5 gives a conceptual overview of the framework as well as embedding the impact assessment algorithm detailed below. The initial systems specification \( S \) has a corresponding assurance case \( A \). These are connected by traceability relation \( R \). System \( S \) is first evolved by changing it to a system specification \( S' \). The difference between these specifications is captured in the relation \( D \). After performing the impact assessment algorithm, the resulting impact set estimate \( A_{RMM} \) and impact kind annotation \( k_{RMM} \) (see Def. 2) are used as guidance by an Assurance Engineer to complete the new assurance case \( A' \) and the corresponding traceability relation \( R' \).

**Impact assessment algorithm.** The impact assessment algorithm used by RMM assumes that potential impact on an assurance case is defined as follows.

**Definition 4 (Potential impact).** An atom in \( A \) is potentially impacted by the change \( D \) iff it is dependent on an atom of \( A \) that mentions the name/identifier of an atom of \( S \) that is itself affected by the change \( D \).

Potential impact means that the atom may be impacted but is not guaranteed to be impacted. For example, it is possible for a claim to be identified as potentially impacted but its truth value happens not to change. However, an atom that does not satisfy this definition is guaranteed to be unimpacted.

The algorithm makes the following assumptions:

**Assumption 1. (RMM Assumptions)**

1.1. Specifications \( S \) and \( S' \) consist of one or more related models defining the systems. We identify the type of these specifications as \( T \).

1.2. Delta \( D \) consists of the three submodels \( C0a \subseteq S' \) and \( C0d \subseteq S \) and \( C0c \subseteq S' \) representing the added atoms, deleted atoms and changed atoms, respectively. Atom addition and deletion can apply to any element, reference or attribute. Atom changes are limited to attribute value changes and changes in the target element of a reference.

1.3 Assurance case \( A \) is considered acceptably correct and complete by the organization developing the system.

1.4 We are provided with a correct model slicer \( \text{Slicer} \) and merge operator \( \text{merge} \) for models of type \( T \). In particular, \( \text{Slicer} \) is assumed to identify all atoms affected by the slicing criterion.

1.5 Traceability relation \( R \) links an atom \( x \in A \) to an atom \( y \in S \) iff \( x \) mentions the name or identifier of \( y \), i.e., \( x \) makes a direct reference to \( y \).

Thus, in Def. 4, checking whether an atom of \( A \) mentions an atom of \( S \) is possible via \( R \) due to Assumption 1.5 and checking whether an atom of \( S \) is affected by \( D \) is possible using \( \text{Slicer} \) due to Assumption 1.4. In addition, checking whether an atom of \( A \) is dependent on another atom of \( A \) is done using the relation \( ACdep \) (see Def. 1).

To determine kinds of impacts on atoms we observe that if an impacted atom of \( A \) mentions an atom of \( S \) that is
deleted, it must be revised because it can no longer be "well-formed". In addition, any other atoms of A that are AC\text{dep} dependent on the revised atom may need revision as well. All other impacted atoms need not be revised but need to be rechecked. Note that this focuses on deletion as the sole cause of revision. Clearly, additions of atoms in S' likely lead to revisions on A as well, but since these are new, the places where such revisions occur cannot be detected by the impact assessment algorithm; thus, these decisions are left for the post-algorithm manual step by the Assurance Engineer.

The algorithm is given in Fig. 4. This a model management algorithm, i.e., it is expressed in terms of standard types of model management operators. In addition, the algorithm is expressed at an abstract level and is parameterized by operators for model type T of the specification S. These operators are given a parameters Slice\textsubscript{T} and Merge\textsubscript{T}. Fig. 5 gives a visual overview of the algorithm embedded in the overall RMM evolution process. The gray circled numbers correspond to line numbers in the algorithm.

In line 1 of the algorithm, the traceability map from S' to A is computed by restricting the original map R using delta D (see Sec. 4.3). Thus R\textsubscript{A}' contains the mappings from all the atoms in S' except the added ones. Lines 2 and 3 use the T-specific slicing operator (Slice\textsubscript{T}) to expand the changed regions to all affected atoms of S and S', respectively. In line 4, these are then traced across the traceability relations to A and merged (see Sec. 4.3) to identify the core subset C2\text{recheck} of A that must be rechecked. In line 5, the core subset C2\text{revise} of A that must be revised is obtained by tracing the set of deleted atoms C0\text{d} across the traceability relation. In lines 6 and 7, these core subsets are expanded to the full impacted subsets using the assurance case slicing operator (see Sec. 4.3) Finally, in lines 8-11, the impact set estimate A\text{imp} and impact kind annotation k\text{imp} are prepared and returned as the output of the algorithm.

**Post-algorithm actions.** After applying the impact assessment algorithm, the Assurance Engineer uses the impact set and impact kind annotation on A as guidance to completing the new assurance case A' for S'. All atoms in A not in A\text{imp} are considered unimpacted and can be reused in A' without change. Those atoms marked 'recheck' must be changed to make them well-formed and then rechecked. The action for an atom marked 'recheck' is dependent on its type. A Claim must be re-evaluated to check that it has an acceptable truth state (e.g., true or affirmed) based on the arguments that support it. For an Evidence element, the procedure for producing the results must be re-performed. For example, if the evidence consists of test results, the test cases must be rerun; for analysis results, the analysis procedure is rerun, etc. In some case the recheck procedure may be automated. For an Argument, its details must be checked to ensure that they are still valid. If the result of the recheck is not acceptable (e.g., new test results fall outside acceptable limits) the Assurance Engineer must assess how to respond. For example, this may mean that the atom must now be revised or it may mean that the system specification requires further changes.

### 4.3 Additional Model Management Operators

Other than the operators Slice\textsubscript{T} and Merge\textsubscript{T}, provided as parameters, the algorithm in Fig. 4 uses four additional model management operators. We describe them declaratively below.

---

**Algorithm: RMM impact assessment**

**Params:** \{Slice\textsubscript{T}, Merge\textsubscript{T}\}

**Input:** initial spec S: T, assurance case A: AC, traceability map R, changed spec S': T, delta D = (C0\text{d}, C0\text{c})

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

1. \(R_A' \leftarrow \text{Restrict}(R, D)\)
2. \(C1dc \leftarrow \text{Slice}_T(S, \text{Merge}_p(C0\text{d}, C0\text{c}))\)
3. \(C1ac \leftarrow \text{Slice}_T(S', \text{Merge}_p(C0\text{d}, C0\text{c}))\)
4. \(C2\text{recheck} \leftarrow \text{Merge}_{AC}(\text{Trace}(R, C1dc), \text{Trace}(R_A', C1ac))\)
5. \(C2\text{revise} \leftarrow \text{Trace}(R, C0\text{d})\)
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. \(A_{\text{imp}} \leftarrow \text{Merge}_{AC}(C3\text{revise}, C3\text{recheck})\)
9. \(k_{\text{imp}}(C3\text{recheck}) \leftarrow \text{recheck'}\)
10. \(k_{\text{imp}}(C3\text{revise}) \leftarrow \text{'revise'}\)
11. \(\text{return } A_{\text{imp}}, k_{\text{imp}}\)

---

**Figure 4:** Algorithm for assessing assurance case impact due to system evolution used in the RMM evolution framework.

**Figure 5:** Conceptual overview of model management based assurance case evolution framework RMM. Numbers in gray circles correspond to the line numbers of the impact assessment algorithm in Fig. 4.

**Definition 5 (Additional Operators).** Let R be the traceability relation between S and A, \(D = (C0\text{d}, C0\text{c})\) be the delta between S and S'. Then

- \(\text{Restrict}(R, D)\) is the relation between S' and A defined as \{(a, c) \mid a \in \text{atoms}_A \wedge c \in (\text{atoms}_{S'} \setminus \text{atoms}_{C0\text{d}}) \wedge R(a, c)\}.

- \(\text{Trace}(R, C)\), where \(C \subseteq \text{atoms}_{S'}\), is the subset of atoms_A defined as \{(a, c) \mid a \in \text{atoms}_A \wedge \exists c \in \text{atoms}_{S'} \cdot R(a, c)\}.

- \(\text{Slice}_{AC}(A, C)\), where \(C \subseteq \text{atoms}_{A}\), is the subset of atoms_A defined as \{a \mid a \in C \wedge AC\text{dep}(a, a')\}.

- \(\text{Merge}_{AC}(A, A')\) is the assurance case \(A' : AC\) defined as \(\text{atoms}_{A'} = \text{atoms}_A \cup \text{atoms}_{A'}\).

\(\text{Restrict}(R, D)\) creates a new traceability relation from S' to A that contains all links in R except those involving deleted atoms. \(\text{Trace}(R, C)\) traverses the traceability relation R to produce the set of atoms linked to atoms in C. \(\text{Slice}_{AC}(A, C)\) expands a subset C of atoms in A to the subset of all atoms dependent on atoms in C through the AC\text{dep} relation defined in Def. 1. \(\text{Merge}_{AC}(A, A')\) produces the assurance case containing the union of atoms in each of
A and A’. Note that in our algorithm we use Merge\textsubscript{AC} to combine submodels of a larger model. If A and A’ overlap, the union includes only one copy of the atoms in the overlap.

5. ALGORITHM ANALYSIS

Def. 3 provided two criteria against which an assurance case impact assessment approach can be evaluated. In this section, we apply these criteria to the impact assessment algorithm in Fig. 4 and briefly discuss the important issue of emergent properties \[17\] in systems and how the algorithm handles this.

5.1 Soundness

As discussed in Sec. 4.2, although the algorithm does find some indirect impacts due to added atoms in C0a, it cannot possibly find them all since the Assurance Engineer is required to assess these. Thus, we limit our soundness claim to evolution due to atom changes and deletions.

PROPOSITION 1. Given a system specification S with an assurance case A and a traceability relation R, and S evolves to a specification S’ with delta D = (C0a, C0d, C0c), the impact assessment algorithm in Fig. 4 is sound as defined in Def. 3 with respect to impacts due to atoms in C0d, C0c.

(Proof sketch) We use Def. 4 as the definition of potential impact and argue by contradiction. Assume the algorithm is not sound. Then there is an atom \(x \in \text{atoms}_{A,x} \notin A_{\text{mrg}}\) and yet it is impacted by some atom \(y \in D\). Since \(x\) is impacted by \(y\), according to Def 4, this means that either (1) \(x\) directly mentions \(y\), (2) \(x\) mentions some other atom \(y’\) in \(S\) that is affected by \(y\), or (3) \(x\) is dependent on another atom \(x’\) in \(A\) that is impacted by \(y\).

In case (1), \(x\) will necessarily be in one of C0c or C0d (added atoms cannot be mentioned in A) and so it will be in C1dc in line 2. Also, if \(a\) mentions \(x\) then it will be in C2recheck or C2revise due to Assumption 1.5 and the use of Trace in lines 4-5. The slicing in lines 6-7 will retain \(x\) since \(AC\text{dep}\) is reflexive and thus \(x \in A_{\text{mrg}}\) is in contradiction to our assumption. Case (2) is similar to case (1) except that we use Assumption 1.4 to ensure that if \(y’\) is affected by \(y\) then it is captured by Slice\textsubscript{2} in lines 2-3. Also, we allow \(x\) to be an added atom in C0a since added atoms can indirectly impact atoms of \(A\) in lines 3-4. Finally in case (3), if \(x’\) is impacted, then \(x’\) must be in C3revise constructed using Slice\textsubscript{AC} in lines 6-7. But since \(AC\text{dep}\) is transitive and \(AC\text{dep}(x, x’)\) holds, Slice\textsubscript{AC} would capture \(x\) in line 6 or 7 and then \(x \in A_{\text{mrg}}\) in line 8 is in contradiction with our assumption. Since we have shown that all cases contradict the assumption that \(x \notin A_{\text{mrg}}\), we conclude that if \(x\) is impacted by \(y\) then \(x \in A_{\text{mrg}}\), and so the algorithm is sound.

5.2 Relative Efficiency

According to Def. 3, an impact assessment approach is more efficient if it reports fewer “false positives”. Because RMM is defined at an abstract level, its efficiency is determined by the information it has available on which to base impact assessments. For example, the algorithm will mark a claim \(x\) to be rechecked if it is \(AC\text{dep}\) dependent on another claim \(x’\). However, if we had access to the particular argument structure connecting the claims, it may be that changing the truth state of \(x’\) does not affect the truth state of \(x\) and so \(x\) should not have been marked for recheck. In Sec. 8 we discuss ways that the efficiency could potentially be improved by utilizing additional information available when the framework is instantiated for particular modeling languages.

5.3 Emergent Properties

An emergent property of \(S\) arises as a result of the integration of parts of \(S\), where no part of the system is directly responsible for it. We consider two cases of emergent properties: system properties \[23\] and feature interactions \[5\].

Consider the following claim for a vehicle: “99.5% of the time, a collision when the vehicle is moving 80kph will not result in a fatality of a passenger.” In an assurance case for the vehicle system, the argument to support this claim might use crash test results as evidence. This property, if true, clearly results from the interaction of the parts of the vehicle rather than from any one part. Evolving the vehicle specification may impact the truth of this claim but some changes would not. For example, a change to the headlight colour would probably have no impact on the truth of the claim. In this scenario, the RMM impact assessment algorithm behaves conservatively – if the vehicle specification evolved in any way, the claim is flagged to be rechecked. This follows because we would expect the specification slicer Slice\textsubscript{2} to identify the whole system “vehicle” to be affected by any change within the specification, and the identifier “vehicle” is mentioned directly in the claim. Thus, although the RMM assessment algorithm sacrifices efficiency by being conservative, it ensures soundness for such system properties.

A feature interaction occurs when using two or more features together results in an unintended behaviour. For example, assume the AutoLight subsystem described above interferes with the AutoOff subsystem responsible for (among other things) turning off the headlights when they are left on after the car is turned off. Even if an assurance case contains separate claims about each subsystem, since they interact, a change to one subsystem could impact the claim of the other. Since the RMM algorithm relies on the slicer Slice\textsubscript{2} to detect dependencies between parts of the specification, the extent to which impacts due to feature interaction are handled correctly depends on the quality of the slicer (see Assumption 1.4). Thus, the algorithm is sound in these situations “up to” the soundness of the slicer.

6. DEMONSTRATION: POWER SLIDING DOOR EXAMPLE

In this section, we demonstrate our general approach for the reuse of assurance case artifacts on an automotive subsystem, namely, a power sliding door system, which is shown to be compliant with part of the ISO 26262 standard. First, we discuss goal refinement as presented in ISO 26262, followed by an instantiation of the framework for specific models of the system and the assurance case. Afterwards, we present the example along with the application of our framework on an evolution scenario.

6.1 Goal Refinement in ISO 26262

ISO 26262 is a relatively recent functional safety standard tailored to meet the particular needs of the automotive industry \[16\]. The standard addresses potential hazards resulting from the malfunctioning of safety-related E/E systems in passenger cars. Undoubtedly, the adoption of ISO 26262 will affect the safety practices of the major car com-
Figure 6: Goal refinement in ISO 26262.

In brief, a SG is a top-level safety requirement that is in place to prevent or mitigate some associated hazards so as to avoid unreasonable risk. FSRs, TSRs, HWSRs, and SWSRs are also safety requirements, derived from SGs, but expressed at different levels of description of the system design. FSRs may be thought of as a technology independent safety requirements derived from SGs, TSRs are technology dependent safety requirements derived from implementation of FSRs, SWSRs and HWSRs are specific safety requirements implemented as part of the software and hardware design. This relationship between SGs, FSRs, TSRs, HWSRs, and SWSRs can be thought of as refinement as follows: SGs are stated, FSRs refine SGs, TSRs refine FSRs, and so on (see Fig. 6). If viewed in this sense, the relationship between SGs, FSRs, TSRs, HWSRs, and SWSRs may be represented as the KAOS tree [6] shown in Fig. 6. Such a KAOS tree makes an assurance case that all safety requirements have been satisfied.

6.2 Instantiating the Framework

For the purpose of the example presented here, we instantiate our general framework such that its input is an initial specification (S) of a system given by a megamodel [9] comprised of a class diagram, a sequence diagram and a relationship between them. This megamodel forms the type (T) of our system specification. The assurance case A for the initial system is given by a KAOS goal tree model (AC), along with traceability to the system megamodel. We are also given an evolution scenario that creates an evolved specification (S′) along with a mapping from the original specification (D). We assume we are given class diagram slice and merge operators, similar to those presented in [22] and [12], respectively. We also assume we are given sequence diagram slice and merge operators similar to those presented in [25] and [29], respectively. Finally, we assume that the slice and merge operators, Slice and Merge, respectively, for the megamodel comprised of the class diagram and sequence diagram and the relationship between them can be computed and are given. One possible way to produce the megamodel slice operator is by first applying the class diagram slice, and

Figure 7: Power sliding door system with redundancy [16].

then, based on the relationship with the sequence diagram, applying the sequence diagram slice to the parts affected.

6.3 Power Sliding Door Example

Consider the example of an automotive subsystem that controls the behaviour of a power sliding door in a car. The system has an Actuator that is triggered on demand by a Driver Switch. This example is presented in Part 10 of the ISO 26262 standard [16]. As per the standard, the power sliding door system is considered an item, with an architecture shown in Fig. 7 (borrowed from Part 10 of ISO 26262). The Driver Switch input is read by a dedicated Electronic Control Unit (ECU), referred to as AC ECU, which powers the Actuator through a dedicated power line. The vehicle equipped with the item is also fitted with an ECU which is able to provide the vehicle speed. This ECU is referred to as VS ECU. The system includes a safety element, namely, a Redundant Switch. Including this element ensures a higher level of integrity for the overall system.

As shown in Fig. 7, the VS ECU provides the AC ECU with the vehicle speed. The AC ECU monitors the driver’s requests, tests if the vehicle speed is less than or equal to 15 km/h, and if so, commands the Actuator. The Redundant Switch is located on the power line between the AC ECU and the Actuator. It switches on if the speed is less than or equal to 15 km/h, and off whenever the speed is greater than 15 km/h. It does this regardless of the state of the power line (its power supply is independent). The Actuator operates only when it is powered.

We present the system design as a combination of a class diagram (see Fig. 8) that describes the various components, their attributes, methods and relationships between them, and a sequence diagram (see Fig. 9) which describes the behaviour of the system. There are three threads running in parallel in the sequence diagram: the top thread describes the behaviour of the Redundant Switch; the middle thread describes the behaviour when the driver requests to open the door, and the bottom thread describes the behaviour when the driver requests to close the door. The traceability between the two models is given implicitly by the sequence diagram referencing objects which are instances of the classes in the class diagram.

Next, following the guidelines of ISO 26262, we consider the following hazard which is obtained via appropriate hazard analysis techniques [16]: HE1: “the activation of the actuator while driving at a speed above 15 km/h, with or without a driver request.”. Then, a safety goal is presented in such a way as to prevent the hazard from occurring: SG1: “Avoid activating the actuator while the vehicle speed is greater than 15 km/h.”.

Once we have a safety goal defined, a set of functional
safety requirements FSR1-5 are put in place to help achieve SG1. FSR1 states that the VS ECU sends accurate vehicle speed information to the AC ECU. Alternatively, this means that the incorrect transmission that the vehicle speed is less than or equal to 15 km/h is prevented. FSR2 states that the AC ECU does not power the Actuator if the vehicle speed is greater than 15 km/h. FSR3 is a requirement that the VS ECU sends accurate vehicle speed information to the Redundant Switch. FSR4 ensures the Redundant Switch is in an open state if the vehicle speed is greater than 15 km/h. And finally, FSR5 ensures that the Actuator operates only when powered by the AC ECU and the Redundant Switch is closed.

Fig. 10 shows a KAOS goal tree that depicts the refinement of SG1 into FSR1-FSR5. This refinement is based on the AND-refinement strategy. For simplicity, we skip the TSRs which are decomposed into HWSRs and SWSRs (see Sec. 6.1), and we go directly from the FSRs to the evidence nodes shown in ellipses. For FSR1, since this is a requirement related to the quality of the vehicle speed sensor, evidence is given by some test results performed on the sensor. For FSR2-5, these can be specified as properties that we can check on the system model using some model-checking tool. Moreover, SG1 and FSR1-FSR5 are all expressed in temporal logic, and it can be proven that \((FSR1 \land FSR2 \land FSR3 \land FSR4 \land FSR5) \implies SG1\). Note that traceability to the system model is given by referencing the parts of the goals that appear in the system model in black font.

### 6.4 Evolution of Power Sliding Door System

Up to this point, we have a description of our system specification \(S\) which is of type “megamodel of class diagram and sequence diagram” \(T\), safety case \(A\) which is of type KAOS (recall that a safety case is a particular kind of assurance case), relationship \(R\) between \(S\) and \(A\), and we are given slice and merge operators for \(T\) \((\text{Slicer}_T\) and \(\text{Merge}_T\)) as discussed in Sec. 6.2. We now describe an evolution scenario and demonstrate how our algorithm works on it.

Consider that the power sliding door system changes in order to decrease its integrity (change in model quality). This could be due to the need to minimize costs and produce a cheaper vehicle. The redundancy is therefore eliminated and the Redundant Switch is removed, as shown in Fig. 11, also borrowed from Part 10 of ISO 26262.

The dynamic VS ECU provides the AC ECU with the vehicle speed. The AC ECU monitors the driver’s requests, tests if the vehicle speed is less than or equal to 15 km/h, and if so commands the Actuator. The Actuator is activated when it is powered.

The parts with labels underlined in each of Fig. 8 and 9 represent the delta \(D\). All the underlined parts of the class diagram and the underlined parts in the top thread of the sequence diagram are to be deleted (C0d); however, the underlined parts in the guards of the bottom two threads of the sequence diagram mean that these guards will be changed (C0c). In this example, no new parts are added in \(S'\), so C0a is empty. \(S'\) is the parts of \(S\) without the underlined components in each of the class and sequence diagram.

Having all of the required parameters \((\text{Slicer}_T, \text{Merge}_T)\) and inputs to the algorithm (initial spec \(S : T\), assurance case \(A : \text{KAOS}\), traceability map \(R\), changed spec \(S' : T\) and delta \(D = (C0a, C0d, C0c)\)), we now demonstrate application of the reuse algorithm presented in Fig. 4.

**Line 1** produces the traceability of the assurance case to the evolved system based on the changes made. This is shown in Fig. 12 by colouring the elements that reference the changed components in black.

**Lines 2-3** use the specific slicers and merge operators for our megamodel as described in Sec. 6.2 to expand the changed regions to all affected elements of \(S\) and \(S'\). This means deleting or changing elements based on deleted \((C0d)\) and changed \((C0c)\) elements, respectively. For example, the removal of the Redundant Switch will impact the behaviour of the Actuator and therefore, its powered and activated states. But the relation between the Actuator and VS ECU is not impacted by the removal of the Redundant Switch, which means the speed reading given to the Actuator is not impacted. This is something we would get from the system-level change impact analysis, and we assume that \(\text{Slice}_T\) is powerful enough to catch that.

**Line 4** identifies the core subset of the original assurance case \(A\) that must be rechecked by tracing from the results of Lines 2-3 back to the original assurance case \(A\). \(C2_{\text{recheck}} = (\text{FSR2, E2})\).

**Line 5** identifies the core subset of original assurance case \(A\) that must be revised by tracing the set \((C0d)\) across the traceability relation between \(S\) and \(A\). \(C2_{\text{revise}} = (\text{FSR3, E3, FSR4, E4, FSR5, E5})\).

**Lines 6-7** expand the core subsets identified in lines 4 and 5 to produce the full rechecked/revised subsets. In this example, SG1 is directly affected by the change and is marked ‘revised’ on line 5. Yet it could be the case that is does not refer to elements being deleted/changed and is marked ‘revised’ in line 6 because it is the parent claim of claims that have been marked ‘revised’, and this would be caught by the assurance case slicer. The results so far imply mean that the set of reusable components is \((\text{FSR1, E1})\).

**Lines 8-11** produce the impact set estimate \(\kappa_{\text{iso}}\) which is an assurance case given by the KAOS tree in Fig. 12, along with the kind annotation \(\delta_{\text{iso}}\) which is represented by the following: checkmark means that claim can be reused safely; circular arrow – that claim should be rechecked; exclamation mark – that claim should be revised.

Finally, the Assurance Engineer will take the result of the algorithm, which is the colour coded goal tree in Fig. 12, and make some decisions to produce an evolved safety case. For example, she may decide that the top level safety goal remains the same, and yet she defines a new set of FSRs that will help achieve this safety goal. FSR1 states that the VS ECU sends the accurate vehicle speed information to the AC.
FSR1: The VS ECU sends the accurate vehicle speed information to the AC ECU.

6 (s.sensed_speed = vehicle_speed)

E1: VS Sensor Accuracy Test Results

Figure 9: Power sliding door system sequence diagram.

E2: Model Checking System Models

E3: Model Checking System Models

E4: Model Checking System Models

E5: Model Checking System Models

Figure 10: Goal tree for system with redundancy.

FSR2: The AC ECU does not power the Actuator if the vehicle speed is greater than 15 km/h.

6 (not(a.powered) and s.sensed_speed > 15)

G(a.powered => vehicle_speed and s.sensed_speed = vehicle_speed)

Figure 11: Power sliding door system without redundancy [16].

ECU (safely reused FSR1 from original system). FSR2 states that the AC ECU does not power the Actuator if the vehicle speed is greater than 15 km/h (rechecked FSR2 from the original system and reused it as it still holds). FSR3 states that the Actuator is activated only when powered by the AC ECU (revised FSR5 and removed part about Redundant Switch). Note that both FSR3 and FSR4 from the original goal tree have been deleted as they were revised and removed since they no longer impact the system or the top level safety goal. A final goal tree for the evolved assurance case is given in Fig. 13. We can then prove that (FSR1 ∧ FSR2 ∧ FSR3) ⊢ SG1. This is compliant with the ISO 26262 refinement guidelines presented in Sec. 6.1, as desired.

7. RELATED WORK

We identify three main categories of related work: work on model evolution, work on modeling of assurances cases, and work on assurance case reuse due to system evolution. We describe them below.

Model evolution. A survey on the evolution of UML models in model-driven software development is presented in [20]. The scenarios that cause a model to change are discussed and they form the basis for system evolution in our approach. Our approach is consistent and complimentary to the existing work on model evolution, including theoretical work on model synchronization [10]. However, we specifically focus on assurance cases as the target of coevolution due to our interest in compliance management. An assurance case in this sense is not a traditional model describing a system specification, but a model of an argument over the system satisfying a property of interest, which differentiates our work from the traditional work on model evolution.

Modeling of assurance cases A variety of methods have been proposed for modeling assurance cases. Goal models and requirements models are used in [15]. [3] presents a formal approach for safety argumentation using KAOS goal models and applies it to a Complex UAV System. The GSN
In the future, we plan to implement the operators and assume that an assurance case can be modelled in a variety of ways as long as it presents the core components – claims, arguments and evidence.

Assurance case reuse due to system evolution. [13] and [19] also approach the problem of safety assessment after design changes. [13] only defines the problem of incremental certification and offers some thoughts on how to address it from the perspective of assurance cases represented in modular GSN diagrams. [19] proposes the notion of patterns for building GSN diagrams. Our model-based approach to assurance case reuse could use the structure of such patterns to identify which parts of the GSN diagram can be reused.

8. CONCLUSION

In this paper, we have presented a generic framework for the reuse of assurance case components due to system evolution. We specified a model management reuse algorithm which uses known model management operators (e.g., slice, merge) and produces a semi-automated solution to the assurance case reuse problem. We evaluated our algorithm and demonstrated its applicability on an automotive subsystem – a power sliding door system.

In the future, we plan to implement the operators and algorithm we presented within a model management tool such as MMINT [8]. We are currently extending MMINT to support model management workflows, defining a slicer-independent API and writing adapters for UML and assurance case model slicers. Depending on the use case, we plan to either use existing slice and merge operators from the literature or define new ones as needed.

We also intend to evaluate our framework on a larger case study, ideally using input from an actual Assurance Engineer. We expect such a study to be done with our industrial partner. We also plan to study the reuse of assurance case components under scenarios other than system evolution. For example, the evolution of standards such as ISO 26262 will impact the assurance case constructed to show compliance of the system to the original standard. Also, safety goal change due to the use of a different hazard analysis technique could occur and will also impact the assurance case. Another scenario is the reuse of assurance case components between similar systems or within a product line of systems. We would also like to study a case where emergent behaviour occurs due to adding new elements to the system [17, 14]. Finally, we would like explore ways to improve our impact assessment approach. One possibility is to find a way to weaken the assumptions made about the power of the provided slice operators and move more of the weight for identifying change impact to our reuse algorithm, thus increasing its application scope. Another possibility is to improve the efficiency of the algorithm by finding a way to exploit the additional knowledge available when the algorithm is instantiated for particular modeling languages.

9. ACKNOWLEDGMENTS

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


