MMINT-A: A Tool for Automated Change Impact Assessment on Assurance Cases

Nick L. S. Fung\textsuperscript{1,2}, Sahar Kokaly\textsuperscript{2}, Alessio Di Sandro\textsuperscript{1}, Rick Salay\textsuperscript{1}, and Marsha Chechik\textsuperscript{1}

\textsuperscript{1} University of Toronto, Toronto, Canada
\{nlsfung,adisandro,rsalay,chechik\}@cs.toronto.edu
\textsuperscript{2} McMaster University, Hamilton, Canada
kokalys@mcmaster.ca

Abstract. Assurance cases are a means to argue about the safety, security, etc., of software systems in critical domains. As systems evolve, their assurance cases can grow in complexity, making them difficult to maintain. In this paper, we present a tool MMINT-A that can, in the context of model-driven development, assess the impact of system changes on their assurance cases. To achieve this, MMINT-A implements an impact assessment algorithm from previous work \cite{7,8} and incorporates a graphical assurance case editor, an annotation mechanism, and two summary tables for the assessment results. We demonstrate the usage of MMINT-A on a Power Sliding Door example from the automotive domain.

Keywords: Assurance Cases · Change Impact Assessment
Tool support

1 Introduction

Assurance cases (AC) are a means to argue about the safety, security, etc., of software systems in critical domains. Since ACs relate to specific system designs, changes to the design will necessitate the corresponding changes to the AC. However, maintaining ACs can be non-trivial. First, they can be very complex; an AC for an air traffic control system may, e.g., comprise over 500 pages and reference some 400 documents \cite{10}. Second, system changes can occur often (especially for software), thus, maintaining ACs can involve frequent and substantial updates.

To facilitate this process, we developed a collection of extensions to the MMINT model management framework \cite{5} to support automated Change Impact Assessment (CIA) on ACs, focusing specifically on the automotive domain. The resulting tool, MMINT-A (which is available at https://github.com/nlsfung/MMINT), identifies how different parts of an AC may be impacted by some given changes to the associated system, whether they can be reused in the updated AC or must revised or rechecked for validity. Thus, the engineer can direct her efforts to reviewing and updating the appropriate parts of the AC.

© Springer Nature Switzerland AG 2018
B. Gallina et al. (Eds.): SAFECOMP 2018 Workshops, LNCS 11094, pp. 60–70, 2018.
https://doi.org/10.1007/978-3-319-99229-7_7
In this paper, we present MMINT-A and describe its features. We outline the MMINT architecture in Sect. 2, describe our extensions to create MMINT-A in Sect. 3, and demonstrate its features using an example from the automotive domain in Sect. 4. We discuss related work in Sect. 5 and conclude in Sect. 6.

2 The MMINT Architecture

As shown in Fig. 1, MMINT (which is available at https://github.com/adisandro/MMINT) is built on top of the Eclipse platform and is designed for managing megamodels, which are defined as collections of related models and are represented as MIDs (Model Interconnection Diagrams). In particular, MMINT supports both the “instance” level, in which models, megamodels and their relations are instantiated and manipulated, as well as the “type” level, in which the necessary metamodels, relation types and model operators are defined.

For example, a metamodel for UML class diagrams can be created on top of the Eclipse platform and plugged into MMINT via the type support runtime layer. The Type MID would then be populated with metadata about the new metamodel, allowing UML class diagrams to be created and manipulated inside megamodels using the MID editor and the ModelRel (model relation) editor. With the workflow editor, new operators can be created by connecting pre-defined model operators into a directed, acyclic network, with the roots being the input models to the workflow and the leaves being the outputs.

![Fig. 1. The architecture of MMINT.](image)

3 Extensions for MMINT-A

MMINT-A comprises a collection of extensions to MMINT for instantiating and operating on ACs. However, since the impact assessment is driven by changes to the system, MMINT-A also requires the availability of appropriate metamodels (and editors) for instantiating the desired system models. Availability of operators, viz., slicers [13], for CIA on individual system models is also assumed since they form part of the overall procedure executed by MMINT-A. Currently, MMINT supports CIA on simplified versions of UML class diagrams.
(CD), sequence diagrams (SD) and state machines (SM), which involved: (1) creating the metamodels using EMF (Eclipse Modeling Framework), (2) creating the corresponding editors using Sirius, (3) implementing each slicer as a Java class, and (4) incorporating them into MMINT by editing their plug-in files.

### 3.1 Assurance Case Metamodel

Our metamodel for ACs (Fig. 2) is derived from the Goal Structuring Notation (GSN) version 1 [1] in which an AC is modelled as a directed acyclic network of six types of elements: goals, strategies, solutions, contexts, justifications and assumptions. The former three form the core of an AC and are connected to each other via “supported-by” relations, with a top level goal as the root and the solutions as leaves, while the latter three are connected to the core via “in-context-of” relations. Each of these elements can also be given a unique identifier and description in accordance to the standard, but we extended it by

![Fig. 2. The AC metamodel in MMINT-A. Concrete and abstract classes are distinguished with black and grey borders, respectively.](image)
adding states to goals and solutions which, although unnecessary for CIA in MMINT-A, allow the user to indicate, respectively, their truth values and the currentness of their evidence as part of the overall change management process.

Furthermore, to capture the CIA results, we introduced annotations to our metamodel, which we previously modelled as comprising three types [8]: Reuse, to indicate that the element is not impacted; Recheck, to indicate that the element may no longer be valid and needs a recheck; and Revise, to indicate that the element’s contents requires changing. However, in MMINT-A, we also distinguish between: (1) recheck content, which indicates that the element’s content may (but not necessarily) require revision, and (2) recheck state, which is applicable to goals and solutions only and indicates that the element’s content, while reusable, may no longer be supported by the underlying sub-goals or evidence.

Focusing on the automotive domain, certain domain-specific features were also incorporated into our metamodel that can be disregarded in general. Specifically, goals are modelled to contain an optional ASIL (automotive safety integrity level) attribute that captures the inherent safety risk of the associated system component and is annotated separately from its goal for CIA. ASILs of sub-goals are generally inherited from their parent goals, but in accordance to ISO 26262 [6], they can be decomposed following certain conditions, which are captured in MMINT-A using a sub-type of strategy (viz., ASILDecompositionStrategy).

3.2 Assurance Case Editor

Figure 3 shows a screenshot of the editor that was implemented on top of Sirius and comprises multiple “views” for creating and visualizing ACs. In the main view (left), ACs are visualized in accordance to the GSN standard but with additional decorations for ASILs and annotations. In particular, ASILs are represented as small, rectangular nodes bordering the goal nodes, while annotations are represented as exclamation marks, circular arrows and check marks to denote Revise, Recheck and Reuse, respectively. The subscripts “C” and “S” are used to disambiguate Recheck Content and Recheck State.

Although compliant with the standard, this graphical representation may not always be the most appropriate. The user may wish to, for example, quickly analyse the amount of impact different changes have on an assurance case or review the source of each annotation. Therefore, to address these use cases, we also created two tabular representations to summarise the results of the CIA. The first table (upper right in Fig. 3) displays the number (and percentage) of each type of node that are annotated for revision, rechecking or reuse, while the second table (lower right in Fig. 3) displays the type of impact for each node and the source of the impact. For example, Fig. 3 shows that goal G1.2.2 must be revised because of a change in the class Redundant Switch.
Fig. 3. Screenshot of the AC editor in *MMINT-A*. The main graphical view is on the left, the statistics table upper right, and the impact trace table lower right.
3.3 Assurance Case Slicers

We make use of two AC slicers based on previous work [8] as part of our overall AC CIA. The Revise Slicer uses rules $V_1$ to $V_4$ in Table 1 to identify elements to be rechecked given an element marked for revision (which applies to the content and not the state of an element). The Recheck Slicer uses the rules $C_1$ and $C_2$ in Table 1 to identify elements to be rechecked given another element marked for rechecking. Note that the Recheck Slicer applies only to state rechecks and that while the Revise Slicer only performs a one-step slice to find direct dependencies of the revised elements, the Recheck Slicer recursively expands a subset of AC elements to include its dependent elements until closure is reached.

Table 1. The assurance case slicer dependency rules.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Element</th>
<th>Dependent element(s) (Annotation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_1$</td>
<td>Goal $G$</td>
<td>All strategies linked to $G$ on either end of the $\text{IsSupportedBy}$ relation (recheck content)</td>
</tr>
</tbody>
</table>
| $V_2$ | Strategy $S$ | 1. All goals that $S$ supports (recheck state)  
2. All goals or solutions that support $S$ (recheck content)  
3. All justifications that are used to justify $S$ (recheck content) |
| $V_3$ | Context $C$ | 1. All goals, strategies and solutions $A$ that introduce $C$ as the context via the $\text{InContextOf}$ relation (recheck content)  
2. All goals, strategies and solutions that inherit $C$ as the context (i.e., all children of $A$) (recheck content) |
| $V_4$ | Solution $S$ | All strategies that $S$ supports (recheck content) |
| $C_1$ | Goal $G$ | All parent goals that are linked to $G$ by the same strategy (recheck content) |
| $C_2$ | Solution $S$ | All goals that are linked to $S$ by the same strategy (recheck content) |

3.4 Change Impact Assessment Algorithm

$\text{MMINT-A}$ implements an AC CIA algorithm based on the one developed in [7,8] which accepts as inputs the original and the updated system megamodel, the set of changes made (i.e., additions, deletions and modifications) as well as the AC and its relation to the megamodel. However, the $\text{MMINT-A}$ implementation assumes that additions can only impact other model elements indirectly via the modifications and deletions required to accommodate them. Thus, the implemented algorithm (see Fig. 4) does not require the added model elements nor the updated system megamodel.

This algorithm is encoded as a $\text{MMINT}$ workflow with 13 model operators. To ensure that the workflow is independent of any specific model types for the input system megamodel, it utilizes higher-level collection-based operators
(particularly, map [14]), enabling it to apply the appropriate operators to the appropriate models in the system megamodel at runtime.

1. Perform CIA on the system megamodel itself.
2. Propagate results to the AC to obtain the elements requiring content recheck.
3. Identify AC elements requiring revision from the deleted system model elements.
4. Apply Revise Slicer on results of step 3.
5. Merge and apply Recheck Slicer on results of steps 2 and 4.
6. Annotate the AC. The results of steps 2, 3 and 5 are marked for content recheck, revision, and state recheck, respectively.

Fig. 4. MMINT-A impact assessment algorithm.

4 Power Sliding Door Example

As part of our evaluation process, we used MMINT-A to recreate a pre-existing case study [8] on the power sliding door (PSD) system presented in ISO 26262 [6] for vehicles. For this case study, the system is modelled using a class diagram (CD) and a sequence diagram (SD) as shown in Fig. 5, and it is associated with

![Class diagram and Sequence diagram for PSD system]

Fig. 5. Models for the PSD system. Recreated in MMINT-A from [8].
Fig. 6. The PSD AC after change impact assessment in MMINT-A.
an AC comprising 22 nodes as shown in Fig. 6. MMINT-A was also applied on a more complex lane management system (LMS) for vehicles which comprises 1 class diagram, 4 sequence diagrams and 4 state machines together with a 74-node AC. However, for illustrative purposes, only the PSD system is presented in this paper.

At a high level, the PSD comprises a driver switch which, when activated by the driver, triggers the AC ECU (actuator electronic control unit) to power an actuator for a door. However, to prevent the door from opening or closing at high speeds (e.g., greater than 15 km/h), the AC ECU is also connected to the VS ECU (vehicle speed ECU) which provides it with the vehicle’s speed. As a backup mechanism, the system also contains a redundant switch that operates independently of the AC ECU to control the actuator, switching on if and only if the speed provided by the VS ECU is sufficiently low.

Accompanying the system models is the AC for the PSD system shown in Fig. 6 (with annotations). The overall goal (G1) for system safety is decomposed into four main subgoals (G1.1 to G1.4), each of which is decomposed further until they are directly supported by the appropriate evidence. The third subgoal (G1.3) illustrates ASIL decomposition, i.e., how introducing an independent redundant switch to the system allows a goal with a high ASIL (C) to be satisfied by subgoals with a lower ASIL in accordance to ISO 26262.

For the case study, we suppose that the redundant switch is removed, and we wish to analyze its potential impact on the AC. To achieve this using MMINT-A, we created a megamodel for the PSD by incorporating the CD and SD models into a MID and adding a relation between them using the ModelRel editor; the appropriate trace links to include in the relation were determined by matching the names of the elements in the class and sequence diagrams. Relations were created similarly between the AC and the system megamodel, all of which formed the inputs to the CIA algorithm, with the original change being the deletion of the redundant switch class and all of its attributes and operations.

The results of executing the workflow in MMINT-A (see Fig. 6) agree with those presented in [8]. For example, Fig. 6 shows that all AC elements that refer directly to the redundant switch must be revised, while any related elements must be rechecked for its content (and/or state) validity. Also, by removing the redundancy mechanism, the ASIL decomposition strategy is no longer valid, thus the ASILs of the corresponding goals must also be revised.

5 Related Work and Discussion

A multitude of tools have been developed over the past two decades to support various aspects of working with ACs [11], many of which can perform CIA like MMINT-A. For example, D-MILS [15] and AutoFocus 3 (ExplicitCase) [3] enable CIA by supporting trace links between system artefacts and the system AC, but unlike MMINT-A, they do not employ slicers to detect indirect impacts of change. Other tools, such as ENTRUST [2] and Evidential Tool Bus (ETB) [4], remove the need for CIA altogether by automatically propagating changes in the
system artefacts to the AC. However, ENTRUST only supports certain changes to the AC, while ECB can only propagate changes to the underlying evidence.

Although the CIA functionality proposed and implemented in MMINT-A can be implemented on top of existing AC tools, these tools are generally highly specialized, making it impractical to adapt them for the automotive domain. On the other hand, before MMINT-A can become a usable tool itself, it must be extended with many “standard” features such as strong support for AC creation and assessment. In fact, unlike D-MILS and AutoFocus 3, MMINT-A can only support trace links to system models; other system artefacts such as natural language documents are not yet incorporated into MMINT.

6 Conclusions and Future Work

In this paper, we presented the features of MMINT-A and demonstrated how it supports the maintenance of ACs, specifically by performing CIA on them. However, since MMINT-A is built on top of the generic MMINT model management framework, it can be easily extended beyond CIA to address other scenarios, including those presented in [9] for regulatory compliance management. In fact, with the appropriate changes to the AC metamodel, MMINT-A can also be applied to non-automotive domains, but because of the focus on models, MMINT-A (and MMINT) may be best suited for model-based software systems only.

In the future, we will use larger case studies, such as that for the lane management system, to better evaluate the effectiveness of the implemented CIA algorithm in MMINT-A. We also aim to, amongst other things, incorporate the improvements suggested in [8], add support for OMG’s Structured Assurance Case Metamodel (SACM [12]) and conduct a usability study with our industrial partner to identify specific barriers in adopting MMINT-A. These may include integration with other tools (e.g., for compliance checking) as well as support for assuring product lines.

Acknowledgements. The work reported in this paper has been funded by General Motors and NSERC Canada. The authors thank Mark Lawford, Alan Wassyng and Tom Maibaum for many useful discussions.

References